**Function:** $A604
Name: DrawCString
Displays a C string in graphics mode
Push: C String (L)
Pull: Nothing
Errors: None



# Chapter 10

# Dialog Boxes

Dialog boxes offer you a chance to communicate with the person using your program. Like the buttons and viewing window on the front of an automatic teller machine, dialog boxes are the most easily understood ways for a computer to display information and obtain input, particularly when

compared to the old-fashioned Yes/No prompts and dreary command line options.

This chapter covers the Dialog Manager and the creation of dialog boxes. Background information is provided initially, with descriptions of the different types of dialog boxes:

- Modal dialog boxes
- Modeless dialog boxes
- Alerts

This chapter also covers the items associated with dialog boxes and all their structures, options, and settings. This is followed by numerous programming examples and explanations. Unlike previous chapters, this chapter does not contain a complete programming example, though you can merge the About. . . dialog box example at the end of this chapter with the MODEL program introduced in Chapter 6.

> Chapter 11, which is about controls, adds a little more information to what's offered here. If you're interested in creating custom dialog boxes with your own controls, it's recommended that you read Chapter 10 first, then Chapter 11.

## Background Information

Dialog boxes are controlled by the Dialog Manager. But actually, more than any other tool set, the Dialog Manager relies on a number of other tool sets to help get the job done. For example, from the previous chapter, you might have read that the Window Manager contributes to the Dialog Manager by drawing the actual dialog box. Also, the Control Manager (covered in the next chapter) helps out by drawing, manipulating, and regulating the controls in a dialog box.

To use dialog boxes in your programs, you'll need to have started the following tool sets:

- Tool Locator
- Memory Manager
- Miscellaneous tool set
- QuickDraw II
- Event Manager





- Window Manager
- Control Manager
- LineEdit tool set

(Also refer to the table of tool set dependencies in Chapter 4.)

It may seem rather strange that the LineEdit tool set is required to use a dialog box. In fact, you cannot display any text in a dialog box unless you've started the LineEdit tool set. The main reason LineEdit is needed is to manipulate text in a text input box (EditLine item).

The text input box, as well as numerous other goodies you can put into a dialog box, are covered in Chapter 11, which deals with controls.

The Dialog Manager is started by a call to the DialogStartUp function and shut down by a call to the DialogShutDown function. The Dialog Manager shares direct page space with the Control Manager, so there's no need to specify direct page space when starting this tool set.

In machine language, the following code can be used to start the Dialog Manager (remember that the above-mentioned tool sets should also have been started):

```
pushword        UserID    ;push the program's User ID
_DialogStartUp            ;No errors possible
```

In C and Pascal:

```
DialogStartUp(UserID);
```

To avoid compile-time errors, C programmers should note that the <dialog.h> header file should be included at the top of your program along with the header files for all the other tool sets that are started up.

To shut down the Dialog Manager, the following routines can be used.

In machine language:

```
_DialogShutDown
```

In C:

```
DialogShutDown( );
```

And in Pascal:

```
DialogShutDown;
```

Once the Dialog Manager is started, your program can display dialog boxes. The dialog boxes can be defined in three ways, using three separate, yet similar, Toolbox calls. Once the dialog box is activated, there are special Dialog Manager calls that monitor the events in the dialog box. All these techniques, including examples of several dialog boxes, are described below.

## Types of Dialog Boxes

As was mentioned earlier in this chapter, there are three types of dialog boxes:

- Modal
- Modeless
- Alert

**Modal.** A modal dialog box is the most common traditional type of dialog box. It's typically a rectangle filled with controls or a message. The dialog box is where a dialogue can take place between the user and the program. A model dialog box allows the user to set or change an option or it can simply display information as in an About... or a Help dialog box.

**Modeless.** The modeless dialog box is the least understood of the three. It's basically a window with dialog controls in it. Unlike the modal dialog box, which is always the foremost window, a modeless dialog box can be placed behind other windows, moved, zoomed, or manipulated like a regular window. Because of this extra activity, the modeless dialog boxes are a little harder to program. Also, their use is vaguely defined, so you won't see them very often.

> Modal? Modeless? How can you remember which one does what?
>
> A good question. Think of a modal dialog as one that puts you in a *mode* where you're essentially forced to interact only with that dialog. A modeless dialog box is one without such restrictions: It's present on the DeskTop, but doesn't force you to interact with it.





**Alert.** The third type of dialog box, the alert, displays a warning and, to varying degrees, a message. Alerts can have OK/Continue or Cancel/Stop buttons in them. The alert dialog boxes are actually specialized forms of modal dialog boxes.

Refer to the Human Interface Guidelines Appendix for more information on the use of the dialog box as well as for design guidelines.

## Creative Overview

Dialog boxes are easy to use. About the hardest thing they require is that you organize your thoughts about what to put into them. Utilizing a combination of tool set functions, the Dialog Manager simplifies the monitoring of dialog box events. Your program acts upon those events and performs whatever actions are necessary.

Dialog boxes, like windows, require tables, locations, pointers, strings—a lot of information. In fact, positioning the controls is the only difficult thing about doing one. You'll spend more time making minor adjustments in the way things are displayed than you will placing them into the dialog box, or debugging logic.

The steps to building a standard, modal dialog box are as follows:

1. Define the dialog box.
2. Place items into the dialog box.
3. Wait for a dialog event.
4. Act on the event (repeat steps 3 and 4 as needed).
5. Close the dialog box when you've finished.

Steps 3 and 4 are repeated as various options in the dialog box are set. According to the Human Interface Guidelines, at least one button in the dialog should be responsible for closing the dialog box and making it go away. Typically, two buttons, OK and Cancel, are used for this purpose.

Actually, a dialog box could contain only a text message such as the famous saying, *Please wait while I initialize.* As soon as the program was ready, it could remove the dialog box and then proceed.

In step 1, the dialog box is defined. It is placed on the screen as a special type of window, in front of all other windows on the screen. There are a number of calls to create the different types of

dialog boxes. In fact, there are three separate Toolbox calls used to create a standard modal dialog box (each is covered later in this chapter).

After the dialog box is created (by whichever method), the Dialog Manager returns a pointer used to further reference the dialog box, just as the Window Manager returns a pointer to a window. The pointer returned by the Dialog Manager is used to place items into that particular dialog box, as well as to remove the dialog box once you've finished with it.

Step 2 is where items are placed into the dialog box. Each item has a position relative to the top left corner of the dialog box (local coordinates), an item description, and a type. The individual characteristics of the items, or controls, placed into a dialog box are covered in the next section.

Steps 3 and 4 are where all the activity takes place. The Dialog Manager has special functions that monitor dialog box activity. These functions take advantage of the TaskMaster and Event Manager to make tracking the events in a dialog box quite simple. When a user selects a particular control, your program can determine which control was selected and take appropriate action.

Once the user has finished with the dialog box (OK or Cancel has been clicked), the dialog box is closed, just like a window. The dialog box can be called up again a number of times by simply repeating these steps. See below for individual examples of how these steps are implemented.

Modal dialog boxes will, without exception, follow the above five steps. Alert boxes are special exceptions. With alerts, the first three steps are combined (most of the work is done internally, by the Toolbox). Alerts are used only to get an immediate yes/no response from a user. Therefore no additional action is taken upon them. They are first displayed; then they get the input and are finally removed so that your program can continue with the action or stop what it's doing. (See the section on alerts below for more information.)

Modeless dialog boxes are handled in a completely different manner. A modeless dialog box is displayed; however, unlike modal dialog boxes and alerts, it need not be acted upon right away. The user can move it behind other windows on the DeskTop, or ignore it completely and go off to do something else. Because of





this, modeless dialog boxes have a special way of handling their events. (Refer to the section on modeless dialog boxes below for additional information.)

## Dialog Box Controls

The things placed into a dialog box are called controls. Buttons are a common type of control, as are radio buttons, check boxes, text input boxes (EditLines), pictures, icons, and even blocks of text.

Every control placed into a dialog box has a special ID number associated with it. It's this value that is monitored by the Dialog Manager's special event-handling routines (step 3 from the previous section). When the user clicks on that control, the ID number is returned for your program to examine. Simple.

Besides assigning an ID number, you also need to define what type of item is placed into your dialog box, where it is placed, whether it's visible, invisible, disabled, and so forth. In all, you need to tell the Dialog Manager seven things in order to place a control into a dialog box (see Table 10-1).

**Table 10-1. Seven Parameters for Placing Controls in Dialog Boxes**

| Name | Value | Meaning |
|---|---|---|
| ItemID | Word | The control's special ID number |
| ItemRect | Rectangle | The control's position inside the dialog |
| ItemType | Word | The type of control: button, text, icon, and so on |
| ItemDescr | Long | A pointer to special information about the control |
| ItemValue | Word | The initial value of a control |
| ItemFlag | Word | Visible/invisible flag, as well as other information |
| ItemColor | Pointer | A table defining the dialog's color |

These items are placed into the dialog box either individually—by using the NewDItem Toolbox call—or all at once—by using a template of information, or record, and using the GetNewDItem or GetNewModalDialog calls.

Individually, each item is described as follows.

**ItemID**. The ItemID is a value assigned to each control in your dialog box. It can be any value in the range $0001–$FFFF. (An Item ID of 0 is possible, but not recommended, because of potential conflicts with certain Toolbox calls.)

An ItemID of 1 is reserved for use by the dialog's default button. Pressing the Return key is considered the same as clicking on the item with an ItemID of 1. Typically, the OK button is given an

ID of 1. Also, if a button has an ID of 1, that button has a double outline.

An ItemID of 2 is reserved for the dialog's Cancel button. Pressing the Escape key is the same as selecting the item in a dialog box with an ID of 2.

Feel free to give the items in your dialog box any number other than 1 and 2 (and 0). A good technique is to give each dialog box an ID in the MSB of the ItemID, then number the controls sequentially starting with 0.

For example, assume your dialog box is given the arbitrary value $0055. Then assign each control in the dialog box (except the OK and Cancel buttons) with IDs of $5500 plus the sequential value of the specific button. Refer to the programming samples below for examples.

> The default button, ItemID $0001, is a good thing to have in any dialog box, especially when you're first writing routines and experimenting. Because pressing the Return key is the same as clicking the default button, if you ever make a terrible formatting mistake (like creating a tall, skinny dialog box with no visible text or controls), you can still press Return to avoid having to reset your computer to start over. This might not exactly be the intent of the default button, but by trial and error, most programmers discover this technique. The authors have become very adept at this.

**ItemRect.** The ItemRect defines the control's position relative to the upper left corner of the dialog box (which is local coordinate 0,0). The ItemRect is defined as four words setting the upper left corner and lower right corner of the control's location as follows:

- Upper Left Y value (MinY)
- Upper Left X value (MinX)
- Lower Right Y value (MaxY)
- Lower Right X value (MaxX)

Any text in your dialog box must fit inside the given rectangle. If you make that rectangle too small, not all the text will be visible. And if you make the rectangle too large, the text might overlay other controls in the dialog box.





With some controls, such as buttons, you need only define the upper left coordinate, using a value of 0 for the lower right coordinate. The lower right values are calculated based on the size of the text inside the control. (This calculation is performed automatically by the Control Manager.) For example,

```
do  12'70.130.0,0'
```

is all right to define the location of a button. The MaxY and MaxX values are set according to the text in the button.

> In machine language and C, the values of a rectangle are given in MinY, MinX, MaxY, MaxX order. But in *TML Pascal*, coordinates use the MinX, MinY, MaxX, MaxY order. Keep this in mind when converting programs between these languages.

**ItemType.** The ItemType parameter describes the type of control. ItemTypes in the following table are listed next to the items they define.

**Table 10-2. ItemTypes and the Control They Describe**

| ItemType | Description | Name | Definition |
|---|---|---|---|
| $000A | Button | ButtonItem | Activator |
| $000B | Check box | CheckItem | Switch |
| $000C | Radio button | RadioItem | Switch |
| $000D | Scroll bar | ScrollBarItem | Special dialog control |
| $000E | User control | UserCtlItem | User-defined |
| $000F | Text | StatText | Characters (up to 255) |
| $0010 | Text (longstat) | LongStatText | Characters (up to 32,767) |
| $0011 | EditLine | EditLine | Input box |
| $0012 | Icon | IconItem | Graphic image |
| $0013 | Picture | PicItem | Graphic image |
| $0014 | User item | UserItem | User-defined |
| $0015 | User control 2 | UserCtlItem2 | User-defined |

Currently, only the above ItemTypes are defined. So, for example, to define a check box in your dialog, you'd specify an item type of $000B (as well as providing the other information indicated in this section).

To disable any item in the dialog box (so that clicking the mouse on that item will not generate a dialog event), logically OR the ItemType with $8000 (which is the same as adding $8000 to

the item value). For example, most text items in a dialog box are disabled, meaning that clicking on them doesn't do anything. To define a disabled text item, the following ItemType can be used:

```
do  i2'$800F'
```

This might also be expressed using equates in machine language (see the examples below), as in

```
do  i2'ItemDisable + StatText'
```

where ItemDisable equals $8000 and StatText equals $000F.

In C, the expression

```
(itemDisable | statText)
```

is equivalent to adding these two items, though more logical.

**ItemDescr.** The ItemDescr is a long word, either a pointer or a handle, depending on the ItemType (see Table 10-3).

**Table 10-3. ItemType Determines What Is Pointed to by ItemDescr**

| ItemType | ItemDescr |
|---|---|
| Picture | Picture's handle |
| Button | Pointer to a string to be placed inside the button |
| Check box | Pointer to the check box's title string |
| Radio button | Pointer to the radio button's title string |
| Scroll bar | Pointer to an action procedure controlling a scroll bar |
| User control | Pointer to the control's action procedure |
| Text | Pointer to the text string |
| Text (longstat) | Pointer to the beginning of the block of text |
| EditLine | Pointer to a text string or buffer |
| Icon | Icon's handle |
| User item | Pointer to a definition procedure |
| User control 2 | Pointer to a parameter block |

All string pointers above indicate the memory location of a Pascal-type string.

**ItemValue.** The ItemValue of a control contains the control's initial value, or 0 in most cases (see Table 10-4).





**Table 10-4. Values Contained in ItemValue**

| ItemType | ItemValue |
|---|---|
| Picture | Pointer to the picture's image |
| Button | Initial value of the control |
| Check box | $0001 to check the box, $0000 for unchecked |
| Radio button | $0001 to fill the button, $0000 to leave it empty |
| Scroll bar | Value passed to the scroll bar's definition procedure |
| Text | Not important |
| Text (longstat) | Number of characters in the text block (up to 32,767) |
| EditLine | Maximum number of characters to be entered (up to 255) |
| Icon | Not important |
| User item | Initial value of the control |
| User control 2 | Initial value of the control |

The value can be examined or changed using the Dialog Manager Toolbox calls GetDItemValue and SetDItemValue. For example, suppose a radio button is to be activated based upon some change in the program. The following routines will change the ItemValue of the radio button.

In machine language:

```
ResetRB  anop
         pha                          ;push one word result space
         pushlong    DialogPtr        ;the pointer to the dialog box
         pushword    #RButton1        ;the ItemID of the radio button
         _GetDItemValue               ;return its value

         pla                          ;test the item's current value
         bne         Go_On            ;if it's already 1, don't change it

         pushword    #$0001           ;the new value for the item (1=on)
         pushlong    DialogPtr
         pushword    #RButton1
         _SetDItemValue               ;set the new value
Go_On    anop
```

*Note:* GetDItemValue and SetDItemValue return an error ($150C) if the ItemID specified does not exist or does not belong to the specified dialog box.

In C:

```
if (!GetDItemValue(DialogPtr, RButton1))
  SetDItemValue(1, DialogPtr, RButton1);
```

In Pascal:

```
IF GetDItemValue(DialogPtr, RButton1) = 0 THEN
  SetDItemValue(1, DialogPtr, RButton1);
```

*Note:* Clicking on a radio button or check box does not automatically activate it. Your program must do that.

The value of the radio button can also be set when the dialog box is initially created. However, the above routines are preferred if the state of the radio button changes. See the COLOR example below. Also, be careful not to confuse changing the ItemValue with making it invisible or disabling it.

**ItemFlag.** The ItemFlag is used mainly by the Control Manager to control certain aspects of some controls—for example, the outline of a button or the orientation of a scroll bar. Refer to Chapter 11 for information on the ItemFlag. For now, setting ItemFlag to 0 in your routines is acceptable.

**ItemColor.** ItemColor is a long word that points to a color table. The color table is used by the Control Manager to change the colors of the item. Normally, this item is set to a long word of 0 and the standard colors are used. Refer to Chapter 11, which deals with controls, for a description of the color table and an example of changing an item's color.

## A Dialog Box

There are three "official" methods for placing a modal dialog box on the screen. The first one is the most complex. It pushes all information about the dialog box on the stack, then calls the Dialog Manager a number of times (once to create the dialog, then one time for each item in the dialog) until everything's finished.

The other two methods use templates of information. These templates merely contain all the data that is pushed to the stack in the first method. However, with templates, only a pointer to a template, or simply to one master template, is pushed to the stack. The Dialog Manager does the rest.

The complex method of creating a dialog is covered in this section, along with important background information. The methods using the templates appear in the following two sections.





To create a dialog box, you must tell the Dialog Manager the following three things:

- The location and size of the dialog box
- Whether the dialog box is visible or not
- A long word value, DRefCon

The DRefCon is a value your program can define for its own use. As with the wRefCon value used by the Window Manager to define a window (see Chapter 9), this value is typically set to 0, but it can be set to any value you'd like.

From the Dialog Manager your program will receive a long-word pointer to the dialog's port, or a long word of 0 if there was an error. This value should be saved for all further references to your dialog box.

Once the dialog is established, you can start placing controls into it. As with creating a dialog box, the controls can be created by pushing their values on the stack and calling a Dialog Manager routine to install them one at a time, or you can use templates to install them all at once.

Simply creating and placing the dialog items does not make them appear in the dialog box. They all suddenly appear the first time you make a call to the ModalDialog function—which is a good thing, because that's what your program will use to handle dialog events.

When the desired controls have been placed into the dialog box, the ModalDialog function handles dialog events, just as the Event Manager or TaskMaster handles desktop events. ModalDialog also initially places all the items into the dialog box. (The items are not visible until ModalDialog is called.)

The ModalDialog function is used only for modal dialog boxes. Modeless dialog boxes and alerts use their own methods for trapping dialog box events. These techniques are discussed in a later section.

ModalDialog waits for the user to click the mouse on a control. When this happens, the ItemID of the control is returned by the ModalDialog function, even for EditLine items. Your program can then take whatever action is necessary.

Once the function of the dialog box is served, close it, removing it from the screen, with the CloseDialog function.

It's important to include some way to close a dialog box. In other words, build in an option for the user to tell the dialog box to go away. It's embarrassing when professional programmers and gurus create magnificent dialog boxes and then realize they have no way of escaping from them.

### Important Pascal Note

At the time of this writing some important *TML Pascal* data types for the Dialog Manager had not been finalized. So, for your programming pleasure, a set of records and data types are listed next. These are all related to working with dialog boxes and alert boxes in Pascal, and they are used throughout the examples in the rest of this book. You can incorporate this information into your programs as needed.

Note that the remainder of this book refers to these types as if they were automatically built into a *TML Pascal* unit symbol file. The definitions of these types won't be shown again.

```
{ TML Pascal Dialog and Alert Type Definitions }
CONST   atItemListLength = 4;
        dtItemListLength = 6;
TYPE    ItemTempPtr = ^ItemTemplate;
        ItemTemplate = PACKED RECORD
                                        ItemID:     Integer;
                                        ItemRect:   Rect;
                                        ItemType:   Integer;
                                        ItemDescr:  Ptr;
                                        ItemValue:  Integer;
                                        ItemFlag:   Integer;
                                        ItemColor:  Ptr;
        END;
DialogTemplate = RECORD
        dtBoundsRect: Rect;
        dtVisible: Boolean;
        dtRefCon: LongInt;
        dtItemList: ARRAY [0..dtItemListLength] OF ItemTempPtr;
END;
AlertTempPtr = ^AlertTemplate;
AlertTemplate = RECORD
        atBoundsRect: Rect;
        atAlertID: Integer;
        atStage1: SignedByte;
```





```
        atStage2: SignedByte;
        atStage3: SignedByte;
        atStage4: SignedByte;
        atItemList: ARRAY [0..atItemListLength] OF ItemTempPtr;
END;
```

Check your version of *TML Pascal* to see whether these types (or similar types) are defined. If they are, the names might be different. (The authors did their best to choose record and field names that seemed the most logical, but they're not clairvoyant.)

### Doing a Dialog, the Long Way

The first Toolbox function used to create a dialog box is NewModalDialog. It receives its information on the stack rather than using a template. The following routines can be used to create a modal dialog box using the NewModalDialog function.

In machine language:

```
LadyDi  pha                             ;long word result space
        pha
        pushlong        #DialogRect     ;rectangle pointer
        pushword        TRUE            ;make dialog visible (TRUE =
                                        ;$8000)
        pea             $0000           ;DRefCon - any value
        pea             $0000
        _NewModalDialog                 ;make the call
        jsr             ErrChk          ;check for errors
        pullong         DialogPtr       ;the dialog pointer
        rts
DialogRect  dc          i2'40,30,100,290'   ;its position and size (320 mode)
```

In C:

```
Rect DialogRect = { 40, 30, 100, 290 };
LadyDi()
{
   DialogPtr = NewModalDialog(&DialogRect, TRUE, NULL);
}
```

In Pascal:

```
PROCEDURE LadyDi;
VAR DialogRect : Rect;
BEGIN
   SetRect(DialogRect, 30, 40, 290, 100);
   DialogPtr := NewModalDialog(DialogRect, TRUE, LongInt(nil));
END;
```

The size and position of the dialog box are specified by the rectangle passed to the NewModalDialog function. According to the Human Interface Guidelines, dialog boxes should be a little higher than screen center and centered left to right. The following machine language equations can be used to center a dialog box. The DHeight and DWidth parameters represent the dialog box's height (Y pixels) and width (X pixels), respectively.

```
DHeight     equ   ??                          ;Your dialog's height goes here
DWidth      equ   ??                          ;Your dialog's width goes here

DialogRect  dc    i2'(190-DHeight)/2'
            dc    i2'(640-DWidth)/2'
            dc    i2'(190-DHeight)/2+DHeight'
            dc    i2'(640-DWidth)/2+DWidth'
```

For a dialog box in the 320 screen mode, change the number 640 above to 320. The value 190 is used for the maximum number of Y pixels to place the dialog box a little above center screen. (It looks awkward when a value of 200 is used.)

This technique can be used in your programs as needed, either as a pointer or as part of a dialog's template (see below). Remember to replace the DHeight and DWidth values in the template with the equates (or values) representing the size of your particular dialog box.

Items placed inside the dialog box are given in local coordinates relative to the upper left corner of the dialog (position 0,0). This allows you to move or resize the dialog box without affecting the internal location of the items.

Items inside a dialog box are placed there by a call to the Dialog Manager's NewDItem function. NewDItem requires the information listed in Table 10-5 for the item you're placing into the dialog.

**Table 10-5. Information Required by NewDItem**

| Parameter | Size | Description |
|---|---|---|
| DialogPtr | Long | A pointer to the dialog box |
| ItemID | Word | The control's ID number |
| ItemRect | Long | A pointer to the control's position |
| ItemType | Word | The type of control |
| ItemDescr | Long | A pointer to special information about the control |
| ItemValue | Word | The control's initial value |
| ItemFlag | Word | Miscellaneous information about the control |
| ItemColor | Long | A pointer to the control's color table |





In the following programming examples, the first control defined is the OK button; the second control, a block of text.

In machine language:

```
ButtonItem  equ   $000A
TextItem    equ   $000F
ItemDisable equ   $8000
TrickDi     pushlong  DialogPtr           ;dialog in which to place this control
            pushword  $0001               ;the ItemID, 1 = default button
            pushlong  #ButtonRect         ;rectangle pointer for the button
            pushword  ButtonItem          ;this item is a button, type $000A
            pushlong  #ButtonText         ;text inside button
            pushword  $0                  ;initial value (not important)
            pushword  $0                  ;itemFlag, zero for default
            pushlong  $0                  ;color table, zero for default
            _NewDItem                     ;make the call
            jsr       ErrChk              ;check for errors

            pushlong  DialogPtr           ;second item: text block
            pushword  $1234               ;ItemID, can be anything
            pushlong  #TextRect
            pushword  TextItem+ItemDisable
            pushlong  #TextText
            pushword  $0
            pushword  $0
            pushlong  $0
            _NewDItem
            jmp       ErrChk

ButtonRect  dc        i2'35,150,0,0'      ;its position in the dialog (relative)
                                          ;ButtonText
str         'OK'      ;button's text
TextRect    dc        i2'10,60,30,240'
TextText    str       'Press the OK button'
```

In C:

```
Rect ButtonRect = { 35, 150, 0, 0 };
Rect TextRect = { 10, 60, 30, 240 };

TrickDi( )
{
   NewDItem(DialogPtr, 1, &ButtonRect, buttonItem,
            "\pOK", 0, 0, NULL);
   ErrChk( );
   NewDItem(DialogPtr, 0x1234, &TextRect, textItem+itemDisable,
            "\pPress the OK button", 0, 0, NULL);
   ErrChk( );
}
```

In Pascal:

```
PROCEDURE TrickDI;
VAR ButtonRect : Rect;
   TextRect : Rect;
   ButtonText : String;
   TextText : String;
BEGIN
   ButtonText := 'OK';
   TextText := 'Press the OK button';
   SetRect(ButtonRect, 150, 35, 0, 0);
   SetRect(TextRect, 60, 10, 240, 30);
   NewDItem(DialogPtr, 1, ButtonRect, ButtonItem,
            @ButtonText, 0, 0, nil);
   ErrChk;
   NewDItem(DialogPtr, $1234, TextRect, StatTextItem+ItemDisable,
            @TextText, 0, 0, nil);
   ErrChk;
END;
```

Once all the controls have been placed in the dialog box, the ModalDialog function is called to monitor dialog events. ModalDialog returns the ItemID of the control selected with the mouse. The following routines incorporate the previous two examples to monitor the pressing of the OK button. When OK is pressed, the dialog box is closed via the CloseDialog function.

In machine language:

```
Wait    pha                         ;one word result space
        pushlong    DialogPtr       ;this dialog
        _ModalDialog                ;make the call

        pla                         ;get results, the ItemID
        cmp         #$1             ;was it OK?
        bne         Wait            ;keep waiting if not OK

        pushlong    DialogPtr       ;close this dialog
        _CloseDialog                ;do it
```

In C:

```
while (ModalDialog(DialogPtr) != 1);
CloseDialog(DialogPtr);
```

In Pascal:

```
REPEAT UNTIL (ModalDialog(DialogPtr) = 1);
CloseDialog(DialogPtr);
```





Once the dialog is closed, the Event Manager/TaskMaster continues monitoring your DeskTop events. The dialog can again be opened to obtain input, adjust settings, or communicate a message, simply by repeating the above steps.

## Making It Easier

The only thing wrong with the routines in the previous section is that they involve a lot of typing (especially in machine language). When you replace the information pushed to the stack with templates of information, the actual code used to create the dialog box becomes easier to read. Also, updating the dialog box is easier because you're changing data templates rather than changing actual program code.

To add a control to a dialog box using templates, the GetNewDItem function is used. GetNewDItem does the same thing as NewDItem, except the information is in a template, and a long pointer to that template is passed to the Toolbox. Refer to Table 10-6 for details about the structure of the template.

**Table 10-6. Structure of GetNewDItem Template**

| Offset | Size | Parameter |
|---|---|---|
| +$00 | Word | ItemID |
| +$02 | Four words (rectangle) | ItemRect |
| +$0A | Word | ItemType |
| +$0C | Long word (pointer) | ItemDescr |
| +$10 | Word | ItemValue |
| +$12 | Word | ItemFlag |
| +$14 | Long word (pointer) | ItemColor |

The following routines are similar to those found in the previous section. They define the same two controls—a button and a block of text—using the GetNewDItem function.

In machine language:

```
ButtonItem  equ         $000A
TextItem    equ         $000F
ItemDisable equ         $8000

PutItems    anop
            pushlong    DialogPtr       ;dialog in which to
                                        ;place this control
            pushlong    #ButtonRec      ;the button's template
            _GetNewDItem
            jer         ErrChk          ;test for errors
```

```
            pushlong     DialogPtr          ;dialog in which to
                                            ;place this control
            pushlong     #TextRec           ;the text's template
            _GetNewDItem
            jmp          ErrChk

ButtonRec   anop                            ;the button's template
            dc           i2'1'              ;itemID, 1
            dc           i2'35,150,0,0'     ;rectangle for the button
            dc           i2'ButtonItem'     ;itemType
            dc           i4'ButtonText'     ;pointer to button's text
            dc           i2'0'              ;initial value (not
                                            ;important)
            dc           i2'0'              ;itemFlag, zero for
                                            ;default
            dc           i4'0'              ;color table, default
ButtonText  str          'OK'               ;button's text

TextRec     anop                            ;the text's template
            dc           i2'$1234'
            dc           i2'10,60,30,240'
            dc           i2'TextItem+itemDisable'
            dc           i4'TextText'
            dc           i2'0'
            dc           i2'0'
            dc           i4'0'

TextText    str          'Press the OK button'
```

In C:

```
ItemTemplate    ButtonRec = {
        1,                          /* itemID = 1 */
        35, 150, 0, 0,              /* rectangle for the button */
        buttonItem,                 /* itemType */
        "\pOK",                     /* button's text */
        0,                          /* itemValue */
        0,                          /* itemFlag */
        NULL                        /* color table, default */
};

ItemTemplate    TextRec = {
        0x1234,
        10, 60, 30, 240,
        textItem + itemDisable,
        "\pPress the OK button",
        0,
        0,
        NULL
};

PutItems( )
{
        GetNewDItem(DialogPtr, &ButtonRec);     ErrChk( );
        GetNewDItem(DialogPtr, &TextRec);       ErrChk( );
}
```

In Pascal:

```
PROCEDURE PutItems;
VAR   ButtonRec :          ItemTemplate;
      TextRec :            ItemTemplate;
      TextText :           String;
      ButtonText :         String;
BEGIN
      TextText                              := 'Press the OK button';
      ButtonText                            := 'OK';
      WITH ButtonRec DO BEGIN
            itemID                          := 1;
            SetRect (itemRect, 150, 35, 0, 0);
            itemType                        := ButtonItem;
            itemDescr                       := @ButtonText;
            itemValue                       := 0;
            itemFlag                        := 0;
            itemColor                       := nil;
      END;
      WITH TextRec DO BEGIN
            itemID                          := $1234;
            SetRect(itemRect, 60, 10, 240, 30);
            itemType                        := itemDisable+StatTextItem;
            itemDescr                       := @TextText;
            itemValue                       := 0;
            itemFlag                        := 0;
            itemColor                       := nil;
      END;
      GetNewDItem(DialogPtr, ButtonRec);    ErrChk;
      GetNewDItem(DialogPtr, TextRec);      ErrChk;
END;
```

The other routines from the previous section, NewModalDialog and ModalDialog (for dialog box event trapping), would still be used as written. The GetNewDItem only aids in the creation of controls.

Do you get the feeling that perhaps you should have started to read this chapter from the end and then worked backwards?

The final step to creating a dialog box easier is just to use one big template for everything—that is, for the dialog box as well as all the controls in the dialog box. This way, creating a dialog box with all its goodies is done with just one Dialog Manager Toolbox call: GetNewModalDialog.

> GetNewModalDialog would seem to be the longest-named Toolbox function. Well, it is. At 17 letters, it ties with FFSoundDoneStatus and TLTextMountVolume. GetNewModalDialog works internally by calling NewModalDialog and then GetNewDItem for each item in the template.

The template used by GetNewModalDialog contains the information listed in Table 10-7. (Note how it also incorporates the templates used by GetNewDItem.)

**Table 10-7. Information Required by GetNewModalDialog Template**

| Offset | Size | Parameter | Description |
|---|---|---|---|
| +$00 | Four words (rectangle) | BoundsRect | Size/location of dialog box |
| +$08 | Word | dtVisible | Visible/invisible flag |
| +$0A | Long word | dtRefCon | Whatever you want |
| +$0E | Long word (pointer) | ItemPtr | First item's template |
| +$12 | Long word (pointer) | ItemPtr | Second item's template (and so on) |
| +$?? | Long word | Terminator | Zero, end of template |

The dialog box's template contains all the information passed to the NewModalDialog function, as well as pointers to the control item's templates used by the GetNewDItem function. The last item in the dialog box's template is a long word of 0 to indicate the end of the template. This way, your dialog box can have a multitude of items (though that's not recommended). See the COLOR example below for a really huge template example.

Incorporating all the information from the previous two sections, the following examples create the dialog box and place all those items into the dialog box. Use the ButtonRec and TextRec data from the examples in the previous section to complete the examples below.





In machine language:

```
PutItems    anop
            pha                                 ;long word result space
            pha
            pushlong    #DialogRec              ;dialog's template
            _GetNewModalDialog                  ;do it all!
            jsr         ErrChk                  ;check for errors
            pullLong    DialogPtr               ;the dialog pointer
            rts
DialogRec   anop                                ;dialog's template
            dc          i2'40,30,100,290'       ;dialog's rectangle
            dc          i2'TRUE'                ;visible flag
            dc          i4'0'                   ;DRefCon - any value
            dc          i4'ButtonRec'           ;first control's template
            dc          i4'TextRec'             ;second control's template
            dc          i4'0'                   ;null terminator
```

In C:

```
DialogTemplate DialogRec = {
                                40, 30, 100, 290,   /* dialog's rectangle */
                                TRUE,               /* visible flag */
                                NULL,               /* dtRefCon */
                                &ButtonRec,         /* first control's template */
                                &TextRec,           /* second control's template */
                                NULL                /* null terminator */
};

PutItems()
{
   DialogPtr = GetNewModalDialog(&DialogRec);
}
```

In Pascal:

```
PROCEDURE
PutItems;
VAR DialogRec : DialogTemplate;
BEGIN
            WITH DialogRec DO BEGIN
                        SetRect(dtBoundsRect, 30, 40, 290, 100);
                        dtVisible       := TRUE;
                        dtRefCon        := LongInt(nil);
                        dtItemList[0]   := @ButtonRec;
                        dtItemList[1]   := @TextRec;
```

```
                    dtItemList[2]     := nil;
            END;
            DialogPtr         := GetNewModalDialog(&DialogRec);
END;
```

Following the above routines, your program should monitor the dialog events with the ModalDialog function and, when finished, close the dialog box with the CloseDialog function. Unfortunately, there are no simple shortcuts for those two calls. (After all, they really are simple themselves.)

The list of ItemTemplate pointers in C is actually an array which has eight elements allocated. If your program has a dialog box that contains more than eight items, you'll have to increase the size of that array to handle more elements. This is done by defining a constant dtItemListLength. It should be placed before the #include <dialog.h> directive at the top of your program—for example:

```
#define dtItemListLength 14   /* define a larger item array */
#include <dialog.h>
```

Pascal programmers need only change the dtItemListLength constant in the CONST section of their programs.

## Alert Boxes

An alert box is a special type of dialog box. It's used to display a message and usually offers two buttons:

- One to go on (OK)
- One to stop whatever action is taking place (Cancel)

Events in alerts are handled by the function that creates the alert. Only one event can be acted upon and then the alert box disappears.

There are four functions to create an alert, each a warning of increasing intensity:

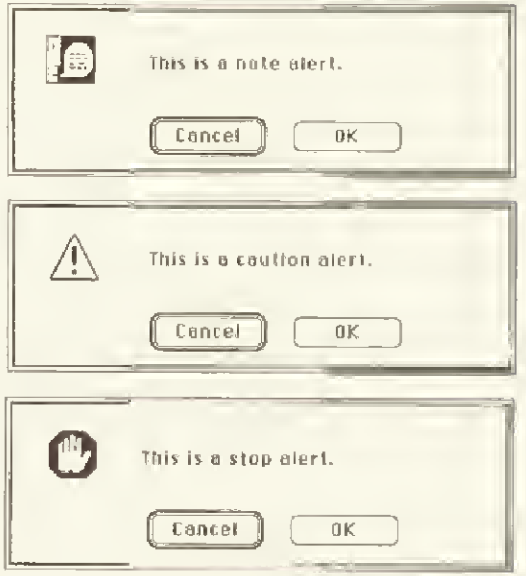
- Alert
- Note alert
- Caution alert
- Stop alert

The note, caution, and stop alerts all have graphic icons associated with them, as seen in Figure 10-1. The basic alert has no graphic. You can define your own icon as the graphic, or just let it go as a text-only alert.





**Figure 10-1. Three Alert Boxes**




The functions to bring up the above alerts are as follows:

| Dialog Manager Function | Type of Alert |
|---|---|
| Alert | Empty alert box (no icon) |
| NoteAlert | Man and cartoon balloon icon |
| CautionAlert | Exclamation point icon |
| StopAlert | Stop sign icon |

These functions are a combination of GetNewModalDialog and ModalDialog. One call to an alert function places all the controls in the specified dialog box (all using one template, as with GetNewModalDialog). Then, ModalDialog is called to monitor the events in the alert box. Control doesn't return from the Toolbox until an item (ItemHit) is chosen.

The only variance among the routines is in the icon drawn (or not drawn) in the upper left corner of the alert box. After an event,

the alert function closes the alert dialog box and returns with the ItemID of the item selected. So, there's really nothing to be done with an alert other than display a message and get a quick response.

Because of the click-and-vanish aspect of alert boxes, they typically only contain text and an OK or Cancel button (or something similar). If you're planning on an alert with more buttons, switches, or controls, you should create a modal dialog box instead.

All the above functions use the following parameters to define an alert:

| Parameter | Size | Description |
|---|---|---|
| AlertTemplatePtr | Long word | Pointer to a template |
| FilterProcPtr | Long word | Pointer to a filter procedure |

The FilterProcPtr points to a user-defined routine to test the events detected by ModalDialog (all dialog events). This way, you can write your own filtering routines, either augmenting or replacing the standard routines used by the Toolbox. Usually, a long word of 0 is specified to use the default routines.

The template pointed to by AlertTemplatePtr contains the information listed in Table 10-8.

**Table 10-8. Information Required by AlertTemplatePtr Template**

| Offset | Size | Parameter | Description |
|---|---|---|---|
| +$00 | Four words (rectangle) | BoundsRect | Size/location of alert |
| +$08 | Word | AlertID | Alert's ID number |
| +$0A | Byte | Stage1 | Alert stage (see below) |
| +$0B | Byte | Stage2 | Alert stage |
| +$0C | Byte | Stage3 | Alert stage |
| +$0D | Byte | Stage4 | Alert stage |
| +$0E | Long word (pointer) | ItemPtr | First item's template |
| +$12 | Long word (pointer) | ItemPtr | Second item's template (and so on) |
| +$?? | Long word | Terminator | Zero, end of template |

The AlertID is simply a unique number identifying the alert box. Its value can be anything.

The alert stages are used to monitor subsequent selection of the same alert box. An alert box is supposed to appear to warn the user of some pending catastrophe. Obviously, the more the alert box tends to pop up in a program, the more careless (or inattentive) the user is. So the differing alert stages can be used to progressively increase the warnings offered by the alert.





**Table 10-9. Bit Values for Alert Stages**

| Bit | Meaning |
|---|---|
| 0 | Number of beeps |
| 1 | Number of beeps |
| 2 | Not used |
| 3 | Not used |
| 4 | Not used |
| 5 | Not used |
| 6 | Sets default button |
| 7 | If set, alert is drawn: if 0, alert is not drawn |

As indicated in Table 10-9 and in Table 10-10, bits 0 and 1 determine the number of beeps made by the alert. The beep sounds before the alert is drawn on the desktop.

**Table 10-10. Beeps Emitted as a Result of Bits Set in Alert Stages**

| Bit | | |
|---|---|---|
| 1 | 0 | Beeps |
| 0 | 0 | None |
| 0 | 1 | One |
| 1 | 0 | Two |
| 1 | 1 | Three |

Bit 6 sets the default button in the dialog. If bit 6 is 0, the default button is ItemID $0001; if bit 6 is 1, the default button is ItemID $0002. (Remember, the default button is selected either with the mouse or by pressing the Return key.)

Bit 7 determines whether the alert is to be drawn or not.

By subtly changing each subsequent alert stage, you can offer an increasingly severe warning each time the same alert appears. Or, you can opt to keep the same alert stage throughout the appearance of your alert dialog. Incidentally, after alert stage 3, alert stage 4 will repeat for each succeeding appearance of the alert.

The following demonstrates four alert stages, each offering a more severe warning than the last:

```
dc  i1'$01'  ;stage one
dc  i1'$81'  ;stage two
dc  i1'$82'  ;stage three
dc  i1'$C3'  ;stage four
```

The first stage simply beeps the speaker once—the alert is not drawn. The second stage beeps the speaker once and the alert is

drawn. In the third stage, the speaker beeps twice before the alert is drawn. In the fourth and all following stages, the speaker beeps three times, the alert is drawn, and the default button is switched. This way, a user who is accustomed to seeing the alert and pressing Return will not automatically continue to select the same option. (He or she will have been foiled—or shocked back into reality, which is the purpose of the alert.)

> A lot of research and study has gone into the way people respond to computers. It seems that no matter how you warn users, no matter how many safeguards and warnings you display, if they are set on doing something, they'll do it, even if that something could lead to catastrophic results.
>
> As an example, it's easy to make an error using the command to reformat a disk on an IBM computer. The only warning offered is a simple *yes/no* prompt. As the accidental formatting of disks increased, the makers of IBM's DOS kept adding safeguards to prevent users from accidentally formatting disks. This still didn't work.
>
> An alert box, on the other hand, has many tricks to continually warn users of what they're about to do. The best is in bit number 6 of the alert stage. This bit switches the default button of an alert. So, if a user is accustomed to seeing the same alert pop up and the natural response is to press Return, you can circumvent that process by switching the way the alert responds to the Return keypress.

As with the GetNewModalDialog function, the alert template ends with a series of pointers to items and controls inside the alert box. A long word of 0 is used to indicate the end of the alert template.

The following example creates a note alert. You can replace the NoteAlert function with either CautionAlert or StopAlert to display a different icon as your own program requires.

In machine language:

```
DoNote   anop
         pea      $0000          ;Result Space (Item ID)
         pushlong #Warning       ;Alert template pointer
         pea      $0000          ;Filter Pointer (use default)
         pea      $0000
```





```
         _NoteAlert
         pla                             ;get Hit Item ID
;evaluation of item hit could be placed here
         rts
Warning  dc       i'50,30,110,290'       ;dialog's rectangle
         dc       i'6374'                ;ID number (unique)
         dc       h'81'                  ;first stage alert
         dc       h'81'                  ;second stage
         dc       h'81'                  ;third
         dc       h'81'                  ;fourth
         dc       i4'item1'              ;First item template
         dc       i4'item2'              ;Second item template
         dc       i4'0000'               ;null terminator
item1    dc       i2'0001'               ;item id
         dc       i2'35,150,60,90'       ;display rectangle
         dc       i2'10'                 ;type = button
         dc       i4'but1'               ;item descriptor
         dc       i2'0'                  ;value of item
         dc       i2'0'                  ;default bit vector
         dc       i4'0'                  ;default color table
item2    dc       i2'6348'               ;item id
         dc       i2'10,60,30,240'       ;display rectangle
         dc       i2'16'                 ;type = text
         dc       i4'msg1'               ;item descriptor
         dc       i2'0'
         dc       i2'0'
         dc       i4'0'
but1     str      'Okay'
msg1     str      'This is a Note Alert'
```

In C:

```
itemTemplate  item1 = {
                                ok,             /* item id */
                                35, 150, 0, 0,  /* item rect */
                                buttonItem,     /* item type */
                                "\pOkay",       /* item text */
                                0, 0, NULL      /* value, bit flag, color
                                                   table*/
};
itemTemplate  item2 = {
                                6348,           /* item id */
                                10, 60, 30, 240,/* item rect */
```

```
                                statText,               /* item type */
                                "\pThis is a note alert",
                                0, 0, NULL              /* value, bit
                                                           flag, and so on*/
};
AlertTemplate  Warning = {
                                50, 30, 110, 290,       /* rectangle */
                                6374,                   /* ID number (unique) */
                                0x81, 0x81,             /* alert stages 1 and 2 */
                                0x81, 0x81,             /* alert stages 3 and 4 */
                                &item1,                 /* first item template */
                                &item2,                 /* second item template */
                                NULL                    /* null terminator */
};
DoNote( )
{
   int ItemHit;
   ItemHit = NoteAlert(&Warning, NULL);
}
```

In Pascal:

```
PROCEDURE DoNote;
VAR  item1 : ItemTemplate;
     item2 : ItemTemplate;
     Warning : AlertTemplate;
     ItemHit : Integer;
     but1 : String;
     msg1 : String;
BEGIN
    but1 := 'Okay';
    msg1 := 'This is a Note Alert';
    WITH item1 DO BEGIN
          ItemID := 1;                          { item id }
          SetRect(ItemRect, 150, 35, 0, 0);     { item rect }
          ItemType := ButtonItem;               { item type }
          ItemDescr := @but1;                   { item text }
          ItemValue := 0;                       { value }
          ItemFlag := 0;                        { bit flag }
          ItemColor := nil;                     { color table }
    END;
    WITH item2 DO BEGIN
          ItemID := 6348;                       { item id }
          SetRect(ItemRect, 60, 10, 240, 30);   { item rect }
          ItemType := StatTextItem;             { item type }
          ItemDescr := @msg1;                   { item text }
          ItemValue := 0;                       { value }
```





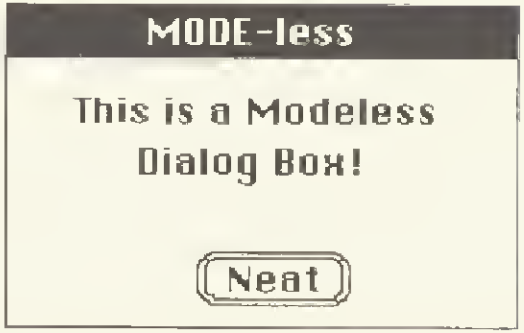
```
          ItemFlag := 0;                        { bit flag }
          ItemColor := nil;                     { color table }
    END;
    WITH Warning DO BEGIN
          SetRect(atBoundsRect, 30, 50, 290, 110);
          atAlertID          := 6374;
          atStage1           := $81;
          atStage2           := $81;
          atStage3           := $81;
          atStage4           := $81;
          atItemList[0]      := em1;  { first item template }
          atItemList[1]      := em2;  { second item template }
          atItemList[3]      := nil;  { null terminator }
    END;
    ItemHit := NoteAlert(@Warning, nil);
END;
```

Remember, in order to use the types AlertTemplate, ItemTemplate, and so forth with older versions of *TML Pascal*, refer to the types defined in the "Important Pascal Notes" section earlier in this chapter.

### A Modeless Dialog Box

Modeless dialog boxes are perhaps the least-understood type of dialog box. Basically, a modeless dialog box is a cross between a window and a dialog box. Unlike most dialog boxes, it can be dragged around, zoomed, and hidden behind other windows—all while still remaining active.

**Figure 10-2. A Modeless Dialog Box**



A good example of a modeless dialog would be a spelling checker in a desktop word processor. The modeless dialog can display a misspelled word and offer a suggestion for the correct spelling. In response, you can edit the text in your document window, then click a Next button inside the modeless dialog box to go to the next misspelling.

Because a modeless dialog box can be active along with everything else on the DeskTop, its events are not handled the same as those in modal dialog boxes.

To handle a modeless dialog box, three separate routines need to be written:

- The routine to create the dialog box
- A modification to the TaskMaster call to detect activity inside the modeless dialog box
- A routine to handle activity inside the modeless dialog box

The routine to create the modeless dialog box works like the routine to create a modal dialog box (but without a template). All the information about the modeless dialog box is specified individually, and then a call is made to the Dialog Manager's function NewModelessDialog.

Items are placed into the dialog box, either via NewDItem or GetNewDItem, and then the function to create the modeless dialog box is complete. The events in the modeless dialog box are then picked up by TaskMaster, so once NewModelessDialog creates the dialog and places it on the screen, your program can go about its business.

In order to monitor the events of the modeless dialog, you need to augment the TaskMaster call in your program's main scanning loop. After the TaskMaster call is made, your program should call the Dialog Manager's IsDialogEvent function. IsDialogEvent returns a logical TRUE value if a modeless dialog event has taken place.

If a modeless dialog event has taken place, your program should branch to a routine to handle activity inside the modeless dialog. That routine calls the DialogSelect function with the ItemID of a control in the modeless dialog box. DialogSelect returns a logical TRUE if that particular item has been selected (see the example below).





The following calls are used to create and manage a modeless dialog box:

| Dialog Manager Function | Action |
|---|---|
| NewModelessDialog | Creates the modeless dialog |
| NewDItem/GetNewDItem | Places items into the modeless dialog |
| IsDialogEvent | Tests for a (modeless) dialog event |
| DialogSelect | Determines which item has been selected |

To create a modeless dialog box, you need to define its size and location, as well as a title, frame description, and the other information you would use when defining a standard window.

In order, NewModelessDialog uses the parameters listed in Table 10-11.

**Table 10-11. Parameters Used with NewModelessDialog**

| Name | Value | Purpose |
|---|---|---|
| DBoundsRectPtr | Long | A pointer to the dialog's rectangle |
| DTitlePtr | Long | A pointer to a Pascal string for the title |
| DBehindPtr | Long | Number of the window the dialog is behind |
| DFlag | Word | Bit pattern describing the dialog's frame |
| DRefCon | Long | Any value: User-defined value, usually 0 |
| DFullSizePtr | Long | A pointer to the size of the dialog when zoomed |

Many of these parameters have similar counterparts in the window record, most notably DBehindPtr, DFlag, and DRefCon.

**DBoundsRectPtr.** DBoundsRectPtr is a long pointer to the address of a rectangle. The rectangle consists of four word values that define the size and location of the modeless dialog using global coordinates. As usual, the values are MinY, MinX, MaxY, and MaxX in that order (unless you're using *TML Pascal*, of course).

**DTitlePtr.** The DTitlePtr is the long address of a Pascal string to be used as the modeless dialog box's title string. If DTitlePtr is a long word of 0, the modeless dialog box does not have a title.

**DBehindPtr.** DBehindPtr acts like wPlane in the window record. It indicates the position of the modeless dialog box in relation to the other windows on the desktop, front to back. DBehindPtr is the value of the window behind which the modeless dialog box is placed. If a value of −1 ($FFFFFFFF) is used, the dialog box is put in front of everything else.

**DFlag.** DFlag is a word-sized bit pattern describing the items in the modeless dialog's frame. The bit positions are exactly the same as they are for wFrame in the window record. Be sure to give your modeless dialog a title bar and don't specify scroll bars (dialog boxes do not have scrolling contents). A common value used for DFlag is $80A0. as seen in the example below.

**DRefCon.** DRefCon, like wRefCon in the window record, can be any long word value you want it to be.

**DFullSizePtr.** DFullSizePtr is a pointer to a rectangle that indicates the size of your dialog box when zoomed. The DFlag option should specify a zoom box in your dialog's title bar in order for the coordinates pointed at by DFullSizePtr to have any effect. A long word of 0 indicates that the zoomed size is the full screen.

The following routines can be used to create a modeless dialog box on your desktop. The modeless dialog box can be called via a pull-down menu or by some other activity in the DeskTop. These routines are written for the 320-mode screen.

In machine language:

```
Modeless    anop
            pushlong    #$0                 ;long word result space
            pushlong    #MDBounds           ;size/location of modeless dialog
                                            ;box
            pushlong    #MDTitle            ;title of modeless dialog box
            pushlong    #$FFFFFFFF          ;place this window in front
            pushword    #$80A0              ;window frame bits
            pushlong    #$0                 ;DRefCon - anything
            pushlong    #$0                 ;Zoomed size (not used)
            _NewModelessDialog
            jsr         ErrChk              ;test for errors
            pullong     ModelessPtr         ;get the pointer

;put an okay button inside the box
            pushlong    ModelessPtr         ;dialog pointer
            pushword    $0001               ;the itemID. 1 = default button
            pushlong    #Button             ;rectangle pointer for the button
            pushword    $000A               ;this item is a button
            pushlong    #Btxt               ;text inside button
            pushword    $0                  ;initial value
            pushword    $0                  ;itemFlag
            pushlong    $0                  ;color table
            _NewDitem
            jsr         ErrChk              ;check for errors
```





```
;put some text in there too:
            pushlong    ModelessPtr         ;dialog pointer
            pushword    $F502               ;itemID
            pushlong    #TextRect
            pushword    $800F               ;text item + item disable
            pushlong    #Text
            pushword    $0
            pushword    $0
            pushlong    $0
            _NewDitem
            jsr         ErrChk
            rts                             ;that's it! All done!
ModelessPtr ds          4                   ;storage for modeless dialog
                                            ;pointer
MDBounds    dc          I2'30,30,100,200'
MDTitle     str         'MODE-less'
Button      dc          I2'40,50,0,0'
Btxt        str         'Next!'
TextRect    dc          I2'10,20,40,160'
Text        dc          I1'endtext-starttext'
starttext   dc          c'This is a Modeless',11'13'
            dc          c' Dialog Box!',11'13'
endtext     anop
```

In C:

```
GrafPortPtr ModelessPtr;
Rect   MDBounds = { 30, 30, 100, 200 };
Rect   BtnRect = { 40, 50, 0, 0 };
Rect   TextRect = { 10, 20, 40, 160 };

Modeless( )
{
    ModelessPtr = NewModelessDialog(&MDBounds,
                                    "\pMODE-less", topMost, 0x80a0, NULL, NULL);
    ErrChk( );
    NewDItem (ModelessPtr, 1, &BtnRect, buttonItem,
              "\pNext!", 0, 0, NULL);
    ErrChk( );
    NewDItem(ModelessPtr, 0xF502, &TextRect, statText + itemDisable,
             "\pThis is a Modeless\r Dialog Box!\r", 0, 0, NULL);
    ErrChk( );
}
```

In Pascal:

```
PROCEDURE Modeless;
```

```
VAR  ModelessPtr : WindowPtr;
     MDBounds : Rect;
     BtnRect : Rect;
     TextRect : Rect;
     Text : String;
     Btxt : String;
BEGIN
   Btxt          := 'Neat!';
   Text          := CONCAT('This is a Modeless', CHR(13),
                           ' Dialog Box!', CHR(13));
   SetRect(MDBounds, 30, 30, 200, 100);
   SetRect(BtnRect, 80, 40, 0, 0);
   SetRect(TextRect, 20, 10, 160, 40);
   ModelessPtr  := NewModelessDialog(MDBounds,'MODE-less',
                           WindowPtr(-1),$80a0, 0, MDBounds);
   ErrChk;
   NewItem(ModelessPtr, 1, BtnRect, ButtonItem, @Btxt, 0, 0, nil);
   ErrChk;
   NewDItem(ModelessPtr, $F502, TextRect, StatTextItem+itemDisable,
                           @Text, 0, 0, nil);
   ErrChk;
End;
```

After the above routines have been called, the modeless dialog box appears on your desktop. The window can be dragged about, just like any other window, but unlike a dialog box, you can pull down menus, open other windows, and perform other activities while the modeless dialog is visible.

To monitor the events in the above modeless dialog, you need to modify your program's main scanning loop with the IsDialogEvent call. IsDialogEvent simply returns a logical TRUE or FALSE if the user has selected something in the modeless dialog. It requires only a pointer to the event record.

The following routine shows how your program's main scan loop can be modified to handle a modeless dialog event.

In machine language:

```
Scan     pha                        ;result Space
         pushword      #$ffff       ;event Mask
         pushlong      #EventRec    ;point to Event Record
         _TaskMaster
         pla                        ;get task code
         beq           Scan         ;if nothing, continue looping
         asl           a            ;double value in A
         tax                        ;put in X for reference
```





```
         jsr           (Table,x)    ;do the appropriate routine
;now, test for a modeless dialog event
         pha                        ;one word result space
         pushlong      #EventRec    ;push the event record
         _IsDialogEvent
         pla                        ;get logical result
         beq           Scan         ;keep looping if FALSE
         jsr           MDEvent      ;otherwise, do the modeless
                                    ;dialog event
         bra           Scan         ;keep scanning for events
```

In C:

```
while (!QFlag){                          /* Wait for an event */
            do {
          Event = TaskMaster(0xffff, &EventRec);
   } while (!Event);
   if (Event == wInMenuBar) DoMenu( );
   if (IsDialogEvent(&EventRec)) MDEvent( );
}
```

In Pascal:

```
REPEAT                          { Wait for an event }
   REPEAT
                      Event := TaskMaster($ffff, EventRec);
   UNTIL Event <> 0;
   IF Event = wInMenuBar THEN DoMenu;
   IF IsDialogEvent(EventRec) THEN MDEvent;
UNTIL QFlag;
```

In the above routines, IsDialogEvent is called after the TaskMaster call. If the result of IsDialogEvent is TRUE, the MDEvent routine is called. MDEvent contains a call to the Dialog Manager's DialogSelect function, the third routine used to monitor events in a modeless dialog box.

When DialogSelect is called, your program can be certain that an event relating to your modeless dialog box has occurred. DialogSelect's job is to determine which control was selected with the mouse so that your program can act accordingly. DialogSelect requires the following parameters:

| Name | Value | Purpose |
|---|---|---|
| TheEventPtr | Long | A pointer to your event record |
| TheDialogPtr | Long | A pointer to the dialog pointer |
| ItemHitPtr | Long | A pointer to an ItemID |

There are quite a few pointers in this function. The actual values are not passed to the DialogSelect function. Only the address of those values is handed down.

The following is an example of a routine to handle the eveots inside a modeless dialog box. It would be called by the previous routine.

In machine language:

```
MDEvent   anop
          pha                        ;one word result space
          pushlong    #EventRec      ;push the event record
          pushlong    #DialogPtr     ;address of dialog pointer
                                     ;storage
          pushlong    #HitItem       ;pointer to hit item
          _DialogSelect
          pla                        ;get logical result
          beq         NoEvent        ;leave it not our hit item
          pushlong    ModelessPtr    ;close this dialog now
          _CloseDialog
NoEvent   rts
DialogPtr ds          4
HitItem   ds          2
```

In C:

```
MDEvent()
{
   GrafPortPtr    DialogPtr;
   Word           ItemHit;
   if (DialogSelect(&EventRec, &DialogPtr, &ItemHit)) {
                  CloseDialog(DialogPtr);
   }
}
```





In Pascal:

```
PROCEDURE MDEvent;
VAR  DialogPtr : WindowPtr;
     ItemHit : Integer;
BEGIN
   IF DialogSelect(EventRec, DialogPtr, ItemHit) THEN
                              CloseDialog(DialogPtr);
END;
```

These routines test for only one item in the dialog box: item 1 (the OK button). If the OK button is clicked, then the DialogSelect function returns a TRUE, and the dialog box is closed. Otherwise, DialogSelect returns FALSE and the program continues.

Multiple DialogSelect calls would be required for a dialog box with more than one selectable control. For each item in the dialog box, a different call to DialogSelect would be made to determine whether that control was activated. (This is because DialogSelect returns only a TRUE or FALSE value, not an ItemHit as with the ModalDialog function and modal dialog boxes.)

## Pretty as an Icon

In this section, and the remaining two sections of this chapter, examples and techniques for modal dialog boxes are listed. You can incorporate these routines into your own dialog boxes.

An icon is a graphic image you can place in your dialog box. It can be a symbol or logo, or it can be a switch to activate some event. However, unlike other types of controls, an icon needs some special adjustment to be placed into a dialog box.

> Actually, anything in a dialog box could be a switch. You simply define that item without adding the item disable to it. The ModalDialog function returns that item's ItemID just as it would return the ItemID of a button, check box, radio button, or any other standard control.

Icons are defined as a series of bytes representing the pixels in the icon's image. They start with a rectangle indicating the size of the icon. The values in the rectangle are

| Offset | Meaning |
|---|---|
| +$00 | Offset of upper left Y coordinate |
| +$02 | Offset of upper left X coordinate |
| +$04 | Height of icon |
| +$06 | Width of icon |

The height of the icon is the number of pixels high the icon will be. The width of the icon is the number of pixels across. For the 640 mode, the width is double that of the 320 mode, even if the icon is of the same size. (The example below is for a 640-mode screen. For a 320-mode screen, the width value would be half of 64, or 32.)

If an icon is to be placed into a dialog box, it must be referenced via a memory handle. This creates a pointer to the icon's data. When you make the NewDItem call, the ItemDescr field becomes the address of that pointer (the address of a pointer is technically known as a handle).

The three programs below are used to create and add an icon to a modal dialog box. (The icon design itself was created for the Living Legends Software company and appears in the About dialogs of most of that company's Apple IIGS software.)

In machine language:

```
Doicon    pushlong    DialogPtr          ;push the dialog pointer
          pushword    #$F804             ;ItemID for the icon
          pushlong    #IconRect          ;rectangle for the icon
          pushword    #IconItem          ;an icon's ItemType, $12
          pushlong    #IconPtr           ;handle (address of pointer)
                                         ;to the icon
          pushword    #0                 ;Item Value
          pushword    #0                 ;Item Flag
          pushlong    #0                 ;color table
          _NewDItem
          jmp         ErrChk

IconRect  dc          i'101,10,117,42'

IconPtr   dc          i4'Icon'           ;pointer to icon's data

Icon      dc          i2'0,0,16,64'      ;size of icon following
          dc          H'FFFFFFFFF0000000FF000000000000FFF'
          dc          H'FFFFFFFFF0FFFFF0FF0FFFFFFFFFF0FFFF'
```





```
          dc          H'FFFFFFF0FFFF0FF0FFFFFFFFFF0FFFFF'
          dc          H'FFFFFF0FFFF0FF0000000000000000F'
          dc          H'FFFFF0FFFF0FFFFFFFFFFFFF0FFFF0FF'
          dc          H'FFFF0FFFF0FFFFFFFFFFFFF0FFFF0FFF'
          dc          H'FFF0FFFF0FFFFFFFFFFFFF0FFFF0FFFF'
          dc          H'FF0FFFF0FFFFFFFFFFFFF0FFFF0FFFFF'
          dc          H'F0FFFF0FFFFFFFFFFFFF0FFFF0FFFFFF'
          dc          H'000000000000000000FF0FFFF0FFFFFF'
          dc          H'FFFF0FFFFFFFFF0FF0FFFF0FFFFFFFF'
          dc          H'FFF0FFFFFFFFFF0FF0FFFF0FFFFFFFF'
          dc          H'FF000000000000FF0000000FFFFFFFFF'
          dc          H'FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF'
          dc          H'FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF'
          dc          H'FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF'
```

In C:

```
#define                    FF      0xff
#define                    F0      0xf0
#define                    0F      0x0f

char      Icon[ ] = { 0, 0, 0, 0, 16, 0, 64, 0,                    /* size */
                      FF,FF,FF,FF,F0,00,00,0F,F0,00,00,00,00,00,0F,FF,  /* data */
                      FF,FF,FF,FF,0F,FF,F0,FF,0F,FF,FF,FF,FF,F0,FF,FF,
                      FF,FF,FF,F0,FF,0F,F0,FF,FF,FF,FF,FF,0F,FF,FF,
                      FF,FF,FF,0F,FF,F0,FF,00,00,00,00,00,00,00,0F,
                      FF,FF,F0,FF,FF,0F,FF,FF,FF,FF,FF,FF,0F,FF,F0,FF,
                      FF,FF,0F,FF,F0,FF,FF,FF,FF,FF,FF,F0,FF,FF,0F,FF,
                      FF,F0,FF,FF,0F,FF,FF,FF,FF,FF,FF,0F,FF,F0,FF,FF,
                      FF,0F,FF,F0,FF,FF,FF,FF,FF,FF,F0,FF,FF,0F,FF,FF,
                      F0,FF,FF,0F,FF,FF,FF,FF,FF,FF,0F,FF,F0,FF,FF,FF,
                      00,00,00,00,00,00,00,00,0F,F0,FF,FF,0F,FF,FF,FF,
                      FF,FF,0F,FF,FF,FF,FF,F0,FF,0F,FF,F0,FF,FF,FF,FF,
                      FF,F0,FF,FF,FF,FF,FF,0F,FF,F0,FF,FF,0F,FF,FF,FF,
                      FF,00,00,00,00,00,00,FF,00,00,00,FF,FF,FF,FF,FF,
                      FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,
                      FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,
                      FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF
};

Rect IconRect = { 101, 10, 117, 42 };

Doicon( )
{
    Ptr IconPtr = Icon;
    NewDItem(DialogPtr, 0xf804, &IconRect, IconItem,
             &IconPtr,0,0,0L);
}
```

In Pascal:

```
PROCEDURE DoIcon;
VAR   IconPtr : Ptr;
      IconRect : Rect;
      Icon : RECORD
      BRect : Rect;
      Data : ARRAY [0..16] OF
                  PACKED ARRAY [1..16] OF Byte;
          END;
BEGIN
   SetRect(IconRect, 10, 101, 42, 117);
   SetRect(Icon.BRect, 0, 0, 64, 16);
   StuffHex(@Icon.Data[0], 'FFFFFFFF000000FF00000000000FFF');
   StuffHex(@Icon.Data[1], 'FFFFFFFF0FFFF0FF0FFFFFFFFFF0FFFF');
   StuffHex(@Icon.Data[2], 'FFFFFFF0FFFF0FF0FFFFFFFFFF0FFFF');
   StuffHex(@Icon.Data[3], 'FFFFFF0FFFF0FF000000000000000F');
   StuffHex(@Icon.Data[4], 'FFFFF0FFFF0FFFFFFFFFFFF0FFFF0FF');
   StuffHex(@Icon.Data[5], 'FFFF0FFFF0FFFFFFFFFFFF0FFFF0FFF');
   StuffHex(@Icon.Data[6], 'FFF0FFFF0FFFFFFFFFFFF0FFFF0FFF');
   StuffHex(@Icon.Data[7], 'FF0FFFF0FFFFFFFFFFFF0FFFF0FFFF');
   StuffHex(@Icon.Data[8], 'F0FFFF0FFFFFFFFFFF0FFFF0FFFFF');
   StuffHex(@Icon.Data[9], '000000000000000FF0FFFF0FFFFFF');
   StuffHex(@Icon.Data[10],'FFFF0FFFFFFFFFF0FF0FFFF0FFFFFFF');
   StuffHex(@Icon.Data[11],'FFF0FFFFFFFFFF0FF0FFFF0FFFFFFF');
   StuffHex(@Icon.Data[12],'FF000000000000FF0000000FFFFFFFF');
   StuffHex(@Icon.Data[13],'FFFFFFFFFFFFFFFFFFFFFFFFFFFFFF');
   StuffHex(@Icon.Data[14],'FFFFFFFFFFFFFFFFFFFFFFFFFFFFFF');
   StuffHex(@Icon.Data[15],'FFFFFFFFFFFFFFFFFFFFFFFFFFFFFF');
   IconPtr := @Icon;
   NewDItem(DialogPtr, $7B04, IconRect, IconItem, @IconPtr, 0, 0, nil);
END;
```

Some touch of compiler magic is required in both the C and Pascal examples. In the C example, to keep the icon data definition as brief as possible, some constants are defined to represent the hexadecimal values $00, $0F, $F0, and $FF. Also note that the icon's size parameters consist of eight characters rather than four word values because of the type of array defined. (A customized structure type could have been used to clean this up, however.) In Pascal, the StuffHex procedure, found in *TML Pascal*'s ConsoleIO unit symbol file, is used to place hexadecimal data into the icon data buffer. Unlike C and machine language, Pascal does not allow you to define an array and have it filled with data at compile time.





## Help

Most DeskTop applications have a feature which provides helpful information about the program. A help dialog box may list special commands used in the program, or explain features that aren't intuitive. Suffice it to say that a help facility is standard equipment for most real-world applications.

This chapter has already presented a number of examples showing how to display text and other information inside a dialog box. But what about changing existing information? For example, what if your help dialog box contained two or more pages of text? How would you switch between screens without creating new dialog boxes for each one?

It's done with two Dialog Manager functions, HideDItem and ShowDItem. When the visible flag is changed on text items, your dialog box can page through them, displaying one screen after another. If your help dialog has three screens of information, the last two are initially hidden, and only the first item is shown. When you go to the next page, perhaps by pressing a Continue button, the first item is hidden and the second item is made visible.

With a little extra tweaking, you could even have buttons specifying Next Page and Previous Page.

## An About. . . Dialog Box

Many chapters in this book have dealt with the MODEL program that was introduced in Chapter 6. This chapter caps off the MODEL program by putting an About. . . dialog box in the Apple menu.

The following code examples can be used to put your standard, run-of-the-mill About. . . dialog box into the MODEL program. This dialog box is rather boring. It only contains text and an OK button. You can add color, icons, or other features to your own dialog boxes. However, when designing a dialog box, you should keep in mind the pointers offered in the Human Interface Guidelines (see Appendix A). While it would be nice simply to drop in the following code as was done in the previous chapter, you will need to make several custom modifications to the MODEL program to facilitate dialog boxes. Most importantly, you'll need to add the Dialog Manager and LineEdit tool sets to the list of tool sets started and shut down by the program.

Once those tool sets have been started, you can replace the empty instruction for About. . . in the MODEL program with the following. To spice it up, you could experiment by adding your own custom icon. (Don't forget to insert the appropriate ShutDown function calls at the end of your program.)

**Program 10-1. Machine Language About. . .**

```
;-------------------------------;
; Apple Menu: About             ;
;-------------------------------;

AboutDialog     equ $F500       ;assign a value to this dialog
ButtonItem      equ $0A         ;id for a button
StatText        equ $0F         ;id for static text
ItemDisable     equ $8000       ;disable an item
About   pea $0000               ;long word result space
        pea $0000
        pushlong #DialogRecord
        _GetNewModalDialog
        jsr ErrChk

        putlong DialogPtr       ;get dialog pointer

;Now wait until the OK button is clicked

Wait    pea $0000               ;result space
        pea $0000               ;filter routine (long pointer)
        pea $0000
        _ModalDialog            ;get dialog events

        pla                     ;get results
        cmp #1                  ;was it the button?
        bne Wait                ;keep waiting if not
```





```
        pushlong DialogPtr      ;we're done, close the dialog
        _CloseDialog

        rts
DialogPtr       ds 4

DialogHeight    equ   60
DialogWidth     equ   400

DialogRecord    anop
                dc i2'(190-DialogHeight)/2'
                dc i2'(640-DialogWidth)/2'
                dc i2'(190-DialogHeight)/2+DialogHeight'
                dc i2'(640-DialogWidth)/2+DialogWidth'
                dc i2'TRUE'
                dc i4'0'
                dc i4 'ButtonRec'
                dc i4 'TextRecord'
                dc i4'0'


ButtonRec       dc i2'1'
                dc i'37,130,0,0'
                dc i2'ButtonItem'
                dc i4 'ButtonText'
                dc i2'0'
                dc i2'0'
                dc i4'0'
ButtonText      str 'OKey Dokey'
```

```
TextRecord  dc i2'AboutDialog+2
            dc i 10,10,80,440
            dc i2 'itemDisable+StatText'
            dc i4'TextString'
            dc i2'0'
            dc i2'0'
            dc i4'0'

TextString  dc i1'endtext-starttext'
starttext   dc c'This is a demonstration program for Advanced '
            dc c'Programming Techniques for the Apple IIGS Toolbox'
            dc i1'13'
endtext     anop
```

### Program 10-2. C About...

```
/*-------------------------------*
 *  Apple menu About             *
 *-------------------------------*/

#define   DialogHeight   80
#define   DialogWidth    400

char le tString[] =
"\pThis is a demonstration program for Advanced\r\
   Programming Techniques for the Apple IIGS Toolbox\r";

ItemTemplate   TextRecord = {
                    2,                  /* item id */
                    10, 10, 80, 440,    /* item rect */
                    itemDisable|statText,   /* item type */
                    textString,         /* item descriptor */
                    0, 0, NULL
```





```
};

ItemTemplate   ButtonRec = {
                    ok,                 /* item id */
                    37, 130, 0, 0,      /* item rect */
                    buttonItem,         /* item type */
                    "\pOkey Dokey",     /* item text */
                    0, 0, NULL          /* value, bit flag, color tbl */
};

DialogTemplate DialogRecord = {
                        (190-DialogHeight)/2,
                        (640-DialogWidth)/2,
                        (190-DialogHeight)/2+DialogHeight,
                        (640-DialogWidth)/2+DialogWidth,
                        TRUE,
                        NULL,
                        &ButtonRec,
                        &TextRecord,
                        NULL
};

About()
{
    GrafPortPtr    DialogPtr;

    DialogPtr = GetNewModalDialog(&DialogRecord);
    while (ModalDialog(NULL) != ok);
    CloseDialog(DialogPtr);
```

**Program 10-3. Pascal About...**

In Pascal:

```
{ +--------------------------------+
  |  Apple Menu: About             |
  +--------------------------------+ }


PROCEDURE About;

VAR     DialogPtr:          WindowPtr;

        TextRecord:         ItemTemplate;

        ButtonRec:          ItemTemplate;

        DialogRecord:       DialogTemplate;

        ButtonText:         String;

        TextString:         String;

BEGIN

    ButtonText := 'Okey Dokey';

    TextString := CONCAT('This is a demonstration program for Advanced Programming',
                         CHR(13), 'Techniques for the Apple IIGS Toolbox',
                         CHR(13));


    WITH ButtonRec DO BEGIN

        ItemID    := 1;                         { item id }

        SetRect   (ItemRect, 130, 37, 0, 0);    { item rect }

        ItemType  := ButtonItem;                { item type }

        ItemDescr := @ButtonText;               { item text }

        ItemValue := 0;                         { value }

        ItemFlag  := 0;                         { bit flag }

        ItemColor := nil;                       { color table }

    END;
```





```
    WITH TextRecord DO BEGIN

        ItemID    := 2;                         { item id }

        SetRect   (ItemRect, 10, 10, 440, 60);  { item rect }

        ItemType  := ItemDisable+StatTextItem;  { item type }

        ItemDescr := @TextString;               { item text }

        ItemValue := 0;                         { value }

        ItemFlag  := 0;                         { bit flag }

        ItemColor := nil;                       { color table }

    END;


    WITH DialogRecord DO BEGIN

        SetRect         (dtBoundsRect, 120, 65, 520, 125);

        dtVisible       := TRUE;

        dtRefCon        := 0;

        dtItemList[0]   := @ButtonRec;

        dtItemList[1]   := @TextRecord;

        dtItemList[2]   := nil;

    END;


    DialogPtr := GetNewModalDialog(@DialogRecord);

    REPEAT UNTIL ModalDialog(nil) = 1;

    CloseDialog(DialogPtr);

END;
```

## Chapter Summary

The following tool set functions were referenced in this chapter.

**Function:** $0215
Name: DialogStartUp
Starts the Dialog Manager
Push: UserID (W)
Pull: Nothing
Errors: None

**Function:** $0315
Name: DialogShutDown
Shuts down the Dialog Manager
Push: Nothing
Pull: Nothing
Errors: None

**Function:** $0A15
Name: NewModalDialog
Creates a modal dialog box
Push: Result Space (L); Rectangle Pointer (L); Visible Flag (W); DRefCon (L)
Pull: Dialog Pointer (L)
Errors: Possible Memory Manager errors

**Function:** $0B15
Name: NewModelessDialog
Creates a modeless dialog box
Push: Result Space (L); Rectangle Pointer (L); Title Pointer (L); Window Level (L); Frame (W); DRefCon (L); Zoomed Rect Pointer (L);
Pull: Dialog Pointer (L)
Errors: Possible Memory Manager errors

**Function:** $0C15
Name: CloseDialog
Removes a dialog from the screen
Push: Dialog Pointer (L)
Pull: Nothing
Errors: Possible Window Manager errors

**Function:** $0D15
Name: NewDItem
Places a control into a dialog box
Push: Dialog Pointer (L); ItemID (W); Rectangle pointer (L); ItemType (W); Item Descriptor (L); ItemValue (W); Item Flag (W); Color Table Pointer (L)
Pull: Nothing
Errors: $150A, $150B

**Function:** $0F15
Name: ModalDialog
Handles events in the frontmost dialog box
Push: Result Space (W); Filter Procedure (L)
Pull: Item Hit (W)
Errors: $150D





**Function:** $1015
Name: IsDialogEvent
Determines whether an event is related to a modeless dialog box
Push: Result Space (W); Event Record Pointer (L)
Pull: Logical Result (W)
Errors: None

**Function:** $1115
Name: DialogSelect
Tests to see whether an item in a modeless dialog box was selected
Push: Result Space (W); Event Record Pointer (L); Dialog Pointer (L); ItemID Pointer (L)
Pull: Logical Result (W)
Errors: None

**Function:** $1715
Name: Alert
Draws an "empty" alert box
Push: Result Space (W); Alert Template (L); Filter Procedure (L)
Pull: Item Hit (W)
Errors: None

**Function:** $1815
Name: StopAlert
Draws an alert box with a stop sign icon
Push: Result Space (W); Alert Template (L); Filter Procedure (L)
Pull: Item Hit (W)
Errors: None

**Function:** $1915
Name: NoteAlert
Draws an alert box with a note icon
Push: Result Space (W); Alert Template (L); Filter Procedure (L)
Pull: Item Hit (W)
Errors: None

**Function:** $1A15
Name: CautionAlert
Draws an alert box with an exclamation point icon
Push: Result Space (W); Alert Template (L); Filter Procedure (L)
Pull: Item Hit (W)
Errors: None

**Function:** $2215
Name: HideDItem
Hides a control in a dialog box, rendering it invisible
Push: Dialog Pointer (L); ItemID (W)
Pull: Nothing
Errors: $150C

**Function:** $2315
Name: ShowDItem
Makes an item or control in a dialog box visible
Push: Dialog Pointer (L); ItemID (W)
Pull: Nothing
Errors: $150C

**Function:** $2E15
Name: GetDItemValue
Returns the value (ItemValue) of a control or item
Push: Result Space (W); Dialog Pointer (L); ItemID (W)
Pull: ItemValue (W)
Errors: $150C

**Function:** $2F15
Name: SetDItemValue
Changes the value of an item, or selects an item
Push: New Item Value (W); Dialog Pointer (L); ItemID (W)
Pull: Nothing
Errors: $150C

**Function:** $3215
Name: GetNewModalDialog
Creates a modal dialog using a template
Push: Result Space (L); Template (L)
Pull: Dialog Pointer (L)
Errors: Possible Memory Manager errors

**Function:** $3315
Name: GetNewDItem
Places an item or control into a dialog box using a template
Push: Dialog Pointer (L); Template (L)
Pull: Nothing
Errors: $150A, $150B

**Window Manager Calls**

**Function:** $0C0E
Name: Desktop
Controls a variety of things dealing with the DeskTop
Push: Result Space (L); Command (W); Parameter (L)
Pull: Result (L)
Errors: None





**Function:** $1D0E
Name: TaskMaster
Returns status of the event queue, updates window events
Push: Result Space (W); Event Mask (W); Event Record (L)
Pull: Extended Event Code (W)
Errors: $0E03

**Memory Manager Calls**

**Function:** $0902
Name: NewHandle
Makes a block of memory available to your program
Push: Result Space (L); Block Size (L); UserID (W); Attributes (W); Address of Block (L)
Pull: Block's Handle (L)
Errors: $0201, $0204, $0207

**Function:** $2002
Name: HLock
Locks and sets a specific handle to a purge level of 0
Push: Handle (L)
Pull: Nothing
Errors: $0206

**Function:** $2802
Name: PtrToHand
Copies a number of bytes from a specific memory address to a handle
Push: Source Address (L); Destination Handle (L); Length (L)
Pull: Nothing
Errors: $0202, $0206

# Chapter 11

# Controls

Controls are things you can put into dialog boxes or windows to perform specific functions. In addition, they have their own identities and allow a user to interact with a program using standards that are maintained in all Apple applications.

The nicest part about controls, like just about everything



else in the Toolbox, is that most of the work relating to them is done for you. You simply define the control, stick it in a window, and your work is done. When you consider that description, a chapter on controls might seem to be useless. Yet, there's a lot of information about controls that doesn't exactly fit under any other rubric. Hence, this chapter is full of information about controls.

This chapter doesn't focus on the Control Manager, but instead concerns itself with the individual controls themselves. The chapter on the Dialog Manager gives dialog boxes a thorough going-over. But much more can be said about controls inside the dialog box. Therefore, this chapter has two areas of concentration:

- The Control Manager
- Controls

The first part of this chapter provides some general information about the Control Manager (one of the more important tool sets). Then the chapter turns to techniques for customizing the standard controls already defined in the Toolbox so that they are best suited to your programs. At the end of this chapter you will find examples of the Control Manager being used to set or change the value of a control.

## The Control Manager

The Control Manager is one of the more important, as well as obscure, tool sets. The following two tool sets rely upon the Control Manager in order to operate properly:

- Window Manager
- Dialog Manager

The reason for this is that both of these tool sets use controls. All the items inside a window—the grow and zoom boxes and the scroll bars—as well as the items in a dialog box are controls. The Control Manager is the tool set whose job it is to manipulate those controls. You can choose from a list of predefined controls; buttons, radio buttons, check boxes, LineEdit boxes, and so on. Or, by using the Control Manager, you can create custom controls to use in your programs.

Many of the functions of the Control Manager are called internally by other tool sets. For example, the Window Manager must

access certain Control Manager functions to place the proper controls into a window. And when you set up a dialog box, it's the Control Manager that handles the intricacies of defining the controls and maintaining their values. As will be seen in a later section, many of the Dialog Manager's functions have similar, corresponding Control Manager functions, some of which are called internally by the Dialog Manager.

Before you start the Control Manager, the following tool sets should already have been started:

- Tool Locator
- Memory Manager
- Miscellaneous tool set
- QuickDraw II
- Event Manager
- Window Manager

To start the Control Manager the CtlStartUp call is made. You'll need to send the Toolbox your program's User ID, and set aside one page ($100 bytes) of direct page space.

In machine language:

```
pushword      UserID     ;push our user id
pushword      DPage      ;push direct page location
_CtlStartUp
jsr           ErrChk     ;check for errors
```

In C:

```
CtlStartUp(UserID, GetDP(0x100)); ErrChk();
```

In Pascal:

```
CtlStartUp(UserID, GetDP($100)); ErrChk;
```

The GetDP call in the C and Pascal examples is described in the MODEL program, illustrated in Chapter 6.

The only error being checked for after the CtlStartUp call is $1001, meaning the Window Manager has not been initialized. So when you're writing applications, it's a good idea to start up the Window Manager before the Control Manager. Also, as is true with all other tool sets, the Control Manager functions better if its allocated direct page space is page-aligned. (See the information on the NewHandle function in Chapter 7 for more information.)





To shut down the Control Manager, a call is made to CtlShutDown.

In machine language:

```
_CtlShutDown
```

In C:

```
CtlShutDown();
```

In Pascal:

```
CtlShutDown;
```

Be careful to shut down the Window Manager before making the above calls. If you're simply shutting down all the tool sets to quit a program, then the order isn't that crucial. Still, it's a good idea to shut down the Window Manager first. You may wonder why this practice is recommended. The reason is that the Window Manager is responsible for disposing of windows (and dialog boxes) containing controls. Therefore it's a good idea to shut it down first. This assures that there are no controls left on the screen when CtlShutDown is called. (CtlShutDown does not remove the controls, so when the Window Manager makes the call to the Control Manager to remove the controls, an error results.)

Shut down tool sets following the reverse of the order in which they were started up.

## Controls

The Control Manager maintains several built-in controls. All the items in a window that manipulate the window are controls. Others managed by the Control Manager include the following items, which you can specify in a dialog box:

- Buttons
- Check boxes
- Radio buttons
- Scroll bars
- Edit lines
- Grow box

For each type of control there is a control record. This record contains information about the control:

- The window to which it belongs
- Pointers to its action procedure
- Pointer to a color table

It also contains information defined by your program when the control was initially put on the screen, or as maintained by the Control Manager as you are manipulating the control.

The following sections detail each type of control. This information is provided to enhance information already presented in Chapter 10. For example, the following sections contain information about certain controls' ItemValue and ItemFlag, and how these values can be manipulated to give your programs their own unique look. Plus, there's information about changing the default color of a control.

The following built-in controls can be specified as part of a dialog box via the NewDItem or GetNewDItem calls of the Dialog Manager. NewDItem specifies each aspect of the control one at a time, whereas GetNewDItem uses a template of values.

In summary, GetNewDItem sets up a call to NewDItem. NewDItem, on the other hand, contacts the Control Manager to set up the control. The Control Manager manipulates the information further and calls NewControl, which actually sets up the control record and assigns the control to a particular window. NewControl may do further initializing depending upon the type of control.

**Push button.** Push buttons always perform some action, or they can activate something. Unlike other controls that can be switched on or off or positioned in some manner, when a push button is clicked by the mouse, it immediately causes something to happen (usually it closes a dialog box).

Table 11-1 shows the items specified when a push button is defined. These items would either be individually specified via the NewDItem function, or using a template with the GetNewDItem function.





**Table 11-1. Items Specified When Push Button Is Defined**

| Name | Size | ButtonItem Value |
|---|---|---|
| ItemID | Word | The button's ID |
| ItemRect | Word | Upper left Y position of the button (MinY) |
| | Word | Upper left X position of the button (MinX) |
| | Word | Usually 0 |
| | Word | Usually 0 |
| ItemType | Word | $000A (10 decimal) |
| ItemDescr | Long | Pointer to string inside the button |
| ItemValue | Word | Always 0 |
| ItemFlag | Word | Determines visibility and type of button |
| ItemColor | Pointer | A table defining the button's color |

*ItemID.* ItemID assigns a unique value to the button. A value of $0001 defines the button as the default button of the dialog box. The default button has a double outline. Pressing Return is the same as clicking the default button.

An ItemID of $0002 defines the default Cancel button, which is equivalent to pressing the Escape key. Other values can be used simply to define a typical push button.

*ItemRect.* The ItemRect of the button defines its location and size relative to the upper left corner of the dialog box (position 0,0). Normally, only the first two words of this rectangle are specified; the last two can be zeros. The Control Manager will fill in the other corner based on the size of the text inside the button.

Later in this chapter, an example of a button is shown with all four values defined. Even though the second two words need not be actual values, the Control Manager will still create a push button (though of a nonstandard size), and will still center the text within that button.

*ItemType.* The ItemType for a button is $000A, or 10 decimal.

> Instead of using a raw number, check your language's support files for predefined symbol names that can greatly improve the readability of your program. For example, when you include the <dialog.h> header file in your C programs, you can use the defined constant called buttonItem rather than the number 0x000a (hex) or 10 (decimal).

*ItemDescr*. ItemDescr is a long-word pointer to the string to be placed inside the button. The string should be rather short, as anything longer than one or two words is considered an essay. When that's the case you should consider whether the button is appropriate. The button's string should start with a count byte (a Pascal string).

*ItemValue*. ItemValue should always be a word of 0. A button does not require an item value.

*ItemFlag*. ItemFlag is a word describing whether the button will be visible or invisible, and it also determines what type of frame the button will have. Only the LSB (lower byte) of this word holds any value; the upper byte should always be 0.

Bit 7 of the ItemFlag word determines the visibility of the button. When bit 7 is set to 1 (a value of $0080), the button is invisible. When bit 7 is reset to 0, the button is visible. There are Dialog Manager and Control Manager functions that will change a button's visibility after it has been created. (Note that there is a difference between a visible button and one that is disabled. See below.)

Bits 0 and 1 of ItemFlag determine the style of the button's frame, or outline. Buttons can have square or round corners, and they can have a double outline or a drop shadow, all depending on how these bits are set.

**Table 11-2. Style of Button's Frame**

| Bit 1 | Bit 0 | Hex Value | Meaning |
|---|---|---|---|
| 0 | 0 | $0000 | Typical round-cornered button |
| 0 | 1 | $0001 | Round-cornered button with double border |
| 1 | 0 | $0002 | Square button |
| 1 | 1 | $0003 | Square button with a drop shadow |

**Figure 11-1. The Four Types of Buttons**






The default button for a dialog box uses a bit pattern of $0001. Other bit patterns for ItemFlag can be used to create different-shaped buttons. However, Apple advises against using the double-border pattern ($0001) on buttons other than the default button.

*ItemColor*. ItemColor is a long-word pointer to a color table for the button. The color table can be used to specify colors other than black and white for the button's parts. For example, the button's text could be green on pink and the button could be gray on blue.

Table 11-3 describes the color table used for a push button (and pointed to by ItemColor).

**Table 11-3. Push Button Color Table**

| Offset | Size | Parameter | Bits 15–8 | Bits 7–4 | Bits 3–0 |
|---|---|---|---|---|---|
| $00 | Word | SimpOutline | 0 | OUT | 0 |
| $02 | Word | SimpNorBack | 0 | BG | 0 |
| $04 | Word | SimpSelBack | 0 | BG | 0 |
| $06 | Word | SimpNorText | 0 | BG | FG |
| $08 | Word | SimpSelText | 0 | BG | FG |

OUT = Outline color
BG = Background color
FG = Foreground color
0 = Always zero

The individual bit positions in each word of the color table are used to specify which colors are used to color each part of the button. In the 320 mode, all four bit positions (7–4 or 3–0) are used to specify one of 16 different colors. In the 640 mode, only bits 4 and 5, or bits 0 and 1, are used to specify color. Be careful to note which values of the word (bitwise) are used and which aren't.

***SimpOutline***. SimpOutline describes the color of the button's outline.

***SimpNorBack***. SimpNorBack is the background color of the button when the button is not being pressed.

***SimpSelBack***. SimpSelBack is the background color of the button when the button is being pressed.

***SimpNorText***. SimpNorText is the color of any text inside the button when the button is not being pressed. The background color of the text is specified in bits 7–4 and the foreground color in bits 3–0.

*SimpSelText*. SimpSelText is the color of any text inside the button when the button is being pressed. The background color of the text is specified in bits 7–4 and the foreground color in bits 3–0.

The following creates a rather interesting colored button (in 320 mode). You might want to include a color table such as this with a program that uses the colorful menu bar example from Chapter 8.

```
ButtonColorT  dc  i2'%0000000000110000'
              dc  i2'%0000000001010000'
              dc  i2'%0000000011010000'
              dc  i2'%0000000001110110'
              dc  i2'%0000000010001001'
```

Notice how similar this is to setting the color table for a window as described in Chapter 9.

**Check box**. A check box represents a condition, either on or off. Clicking in a check box doesn't automatically turn it on, or activate it. Instead, its ItemValue must be changed either through the SetDItemValue call in the Dialog Manager, or via Control Manager calls as outlined in a later section of this chapter. (This was covered briefly in the previous chapter.) When you click the mouse in a check box, it should become checked if it wasn't already, or it should become unchecked if it was. This logic is supplied by your program.

Check boxes have a line of text beside them. Unlike static text items, the text by a check box is defined along with other attributes of the check box. Therefore, the position of the check box on the screen should account for any text just to the right of it.

Table 11-4 shows the values used to define a check box:

**Table 11-4. Values Used to Define a Check Box**

| Name | Size | CheckItem Value |
|---|---|---|
| ItemID | Word | The check box's ID |
| ItemRect | Word | Upper left Y position of the check box (MinY) |
| | Word | Upper left X position of the check box (MinX) |
| | Word | Zero |
| | Word | Zero |
| ItemType | Word | $000B (11 decimal) |
| ItemDescr | Long | Pointer to check box's title string |
| ItemValue | Word | $0000 for open, any other value for selected |
| ItemFlag | Word | Determines visibility |
| ItemColor | Pointer | A table defining the box's color |





*ItemID*. The ItemID of a check box can be any value used to identify the checkbox uniquely. You could specify an ItemID of $0001 or $0002; it isn't recommended, however. This would clash with the rules set down in Apple's Human Interface Guidelines. Only a push button should be the default button in a dialog box, so only a push button should have an ItemID of $0001 or $0002.

*ItemRect*. ItemRect, like a button, defines the location of the check box relative to the upper left corner of the dialog box. Any text appearing next to the check box will be to the right of the check box. As with a button, keep the text brief.

*ItemType*. The ItemType of a check box is $000B, or 11 decimal.

*ItemDescr*. ItemDescr is a long-word pointer to the string appearing next to the check box. The string should start with a count byte.

*ItemValue*. ItemValue indicates the initial value of the check box. If ItemValue is 0, the check box is empty, or unchecked. If ItemValue is any nonzero value, the check box is checked, indicating that whatever state the check box is monitoring is presently selected, or active.

*ItemFlag*. A check box's ItemFlag holds the same meaning that it does for a push button: It determines whether the check box will be visible or invisible. A value of $0080 means the check box will be invisible, while a value of $0000 means the check box will be visible.

*ItemColor*. ItemColor is a long-word pointer to a color table for the check box. Table 11-5 describes the items in a check box's color table.

**Table 11-5. Items in Check Box's Color Table**

| Offset | Size | Parameter | Bits | | |
|---|---|---|---|---|---|
| | | | 15–8 | 7–4 | 3–0 |
| $00 | Word | CheckReserved | 0 | 0 | 0 |
| $02 | Word | CheckNorColor | 0 | BG | FG |
| $04 | Word | CheckSelColor | 0 | BG | FG |
| $06 | Word | CheckTitleColor | 0 | BG | FG |

BG = Background color
FG = Foreground color
0 = Always zero

The same information for a push button's color table (regarding bit positions) holds true for this and all succeeding color tables. Remember that the 320 mode is much more colorful than the 640 mode.

*CheckReserved.* CheckReserved should be a word of 0. Presumably Apple has something clever in mind for this value and just won't let us know what it means.

*CheckNorColor.* CheckNorColor is the color of the check box when it's not highlighted or selected.

*CheckSelColor.* CheckSelColor is the color of the check box when it's highlighted or selected. An example of color usage would be to specify bits 7–4 to show a different color (say, red) for a selected check box.

*CheckTitleColor.* CheckTitleColor is the background and foreground color of the check box's title string at all times. (The title does not change as the box changes.)

**Radio button.** Radio buttons are among the most useful types of controls. Yet they are also easily misunderstood. With radio buttons, only one in a series can be selected at a time—and one of the series must be on. Figure 11-2 gives an example of a good use for radio buttons.

**Figure 11-2. Row of Three Radio Buttons: Up, Down, and From Top**






> Why call them radio buttons? The analogy Apple gives is that of an old car radio. The buttons on the radio were used to switch from one preselected radio station to another. Only one of the buttons could be down at a time—you couldn't listen to more than one station. When you pushed one in, any other button that was pressed in would be automatically released.
>
> Radio buttons should be used in an application when one of several options must be selected, but not more than one. If it's possible to choose more than one option, check boxes should be used.

You can specify which radio button is to be on when the dialog box is created. However, as with other items in a dialog box, further manipulation of the radio buttons is up to your program. (Refer to the COLOR program from Chapter 10 for a good example of radio button manipulation.)

Table 11-6 shows the values used to define a radio button.

**Table 11-6. Values Used to Define Radio Buttons**

| Name | Size | RadioItem Value |
|---|---|---|
| ItemID | Word | The radio button's ID |
| ItemRect | Word | Upper left Y position of the button (MinY) |
| | Word | Upper left X position of the button (MinX) |
| | Word | Zero |
| | Word | Zero |
| ItemType | Word | $000C (12 decimal) |
| ItemDescr | Long | Pointer to radio button's title string |
| ItemValue | Word | $0000 for open, any other value for selected |
| ItemFlag | Word | Determines visibility and family number |
| ItemColor | Pointer | A table defining the button's color |

*ItemID.* The ItemID of a radio button, as with a check box, can be any value except $0001 or $0002. A *family number* can be given to a radio button via its ItemFlag value. This family number is used to group radio buttons according to their function, and to ensure that only one radio button within a particular family is on at a time. (The Control Manager will actually prevent you from activating more than one radio button at a time. See the ItemFlag description below.)

*ItemRect.* ItemRect defines the radio button's location relative to the upper left corner of the dialog box. Any text appearing next to the radio button will be to its right.

*ItemType.* The radio button ItemType is $000C, or 12 decimal.

*ItemDescr.* ItemDescr is a long-word pointer to a Pascal string to appear next to the radio button.

*ItemValue.* ItemValue indicates the initial value of the radio button. As with a check box, when ItemValue is 0, the radio button is unselected, and when ItemValue is any nonzero value, the radio button is highlighted.

*ItemFlag.* ItemFlag determines the visibility of the radio button as well as its family number. Bit 7 of the ItemFlag word determines visibility. When this bit is set to 1, the radio button is invisible; when bit 7 is reset to 0, the radio button is visible. The remainder of the bits in this word (bits 6–0) specify the family number of the button. Values in the range $0000–$007F can be used for up to 128 family numbers.

*ItemColor.* ItemColor is a long word pointer to a color table for the radio button.

**Table 11-7. Meaning of Bits Within ItemColor**

| Offset | Size | Parameter | Bits | | |
|---|---|---|---|---|---|
| | | | 15–8 | 7–4 | 3–0 |
| $00 | Word | RadioReserved | 0 | 0 | 0 |
| $02 | Word | RadioNorColor | 0 | BG | FG |
| $04 | Word | RadioSelColor | 0 | BG | FG |
| $06 | Word | RadioTitleColor | 0 | BG | FG |

BG = Background color
FG = Foreground color
0 = Always zero

***RadioReserved.*** RadioReserved is a word of 0, reserved for some future date. Perhaps Apple will design a three-dimensional radio button selected with this value.

***RadioNorColor.*** RadioNorColor is the color of the radio button when it's not highlighted or selected.

***RadioSelColor.*** RadioSelColor is the color of the radio button when it is highlighted or selected.

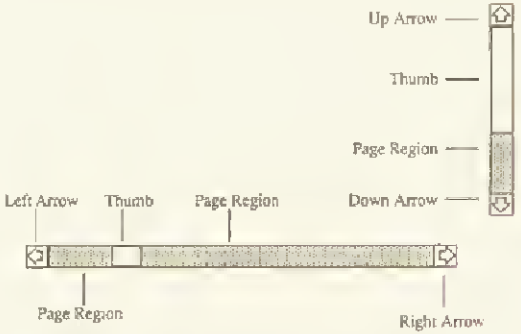
***RadioTitleColor.*** RadioTitleColor is the background and foreground color of the radio button's title string.





**Scroll bar.** You may not think of scroll bars as controls, but they are. They're just like buttons, check boxes, and radio buttons. They're usually used with windows. However, they can be used for other purposes if you know how to manipulate them.

**Figure 11-3. Diagram of Scroll Bar with Associated Terms**




The scroll bar is the most complex type of control you can define. The Window Manager uses scroll bars in windows to scroll an area of data. However, if you want to put a scroll bar into a dialog box just to see what it's like, you'll need to know the information provided by Table 11-8.

**Table 11-8. Information Required to Define a Scroll Bar**

| Name | Size | ScrollBarItem Value |
|---|---|---|
| ItemID | Word | The scroll bar's ID |
| ItemRect | Word | Upper left Y position of the scroll bar (MinY) |
| | Word | Upper left X position of the scroll bar (MinX) |
| | Word | Lower right Y position of the scroll bar (MaxY) |
| | Word | Lower right X position of the scroll bar (MaxX) |
| ItemType | Word | $000D (13 decimal) |
| ItemDescr | Long | Zero, or a pointer to an action procedure |
| ItemValue | Word | Data size minus view size (greater than 0) |
| ItemFlag | Word | Determines visibility and scroll bar items |
| ItemColor | Pointer | A table defining the scroll bar's color |

*ItemID.* ItemID is a value used to identify the scroll bar.

*ItemRect.* ItemRect defines the scroll bar's location in the dialog

box (or window), relative to the dialog box's upper left corner (local coordinates). The two words indicating the lower right corner of the scroll bar take on significance here and must be specified. Together the four word values create the rectangle into which the Control Manager will squeeze the scroll bar.

By adjusting the corner positions of the scroll bar, you can have a very skinny scroll bar, or one that's terribly fat. Because a scroll bar is a predefined control, you can subtly change the way it looks to use it as a custom control in your programs.

*ItemType.* The ItemType of a scroll bar is $000D, or 13 decimal.

*ItemDescr.* ItemDescr is the long-word address of a scroll bar action procedure used to control the scroll bar. A long word of 0 can be used to specify the default procedure.

*ItemValue.* ItemValue indicates the position of the *thumb* in the scroll bar. The higher the value, the further along in position the thumb will be (with the origin at the top or far left of the scroll bar, depending upon the scroll bar's orientation).

*ItemFlag.* ItemFlag determines the visibility of the scroll bar, as well as the orientation of the scroll bar and what types of arrows it will have. (The thumb and page regions of the scroll bar are included standard, but the up/down or right/left arrows are considered optional.) As with other ItemFlag values, only bits 7 through 0 hold any significant value in this word. All other bits should be reset to 0.

Table 11-9 shows the meanings of the bit positions in a scroll bar's ItemFlag.

**Table 11-9. Meaning of Bit Positions in Scroll Bar's ItemFlag**

| Bit | Meaning if Set |
|---|---|
| 7 | Scroll bar is invisible |
| 6 | Nothing (should always be 0) |
| 5 | Nothing (should always be 0) |
| 4 | Scroll bar is horizontal (right to left) |
| 3 | Scroll bar will have a right arrow |
| 2 | Scroll bar will have a left arrow |
| 1 | Scroll bar will have a down arrow |
| 0 | Scroll bar will have an up arrow |

If bit 4 above is reset to 0, the scroll bar will be vertical, or up and down.





You can specify arrows either in one or both directions (up/down, left/right) for your scroll bar. It's possible to specify a left/right arrow with an up/down scroll bar, even though it's wrong. Your program will not crash, but the scroll bar will be updated improperly and your dialog box will fill with random graphics. In other words, it's ill-advised.

So, to specify a full-on vertical scroll bar with both arrows, an ItemFlag of $0003 is used. For a full-on horizontal scroll bar, an ItemFlag of $001C can be used.

*ItemColor.* ItemColor is a long word pointer to the scroll bar's color table as shown below.

**Table 11-10. Meaning of Bits Within ItemColor**

| Offset | Size | Parameter | Bits | | |
|---|---|---|---|---|---|
| | | | 15-8 | 7-4 | 3-0 |
| $00 | Word | ScrollOutline | 0 | OUT | 0 |
| $02 | Word | ArrowNorColor | 0 | BG | FG |
| $04 | Word | ArrowSelColor | 0 | BG | FG |
| $06 | Word | ArrowBackColor | 0 | BG | 0 |
| $08 | Word | ThumbNorColor | 0 | BG | 0 |
| $0A | Word | ScrollReserved | 0 | 0 | 0 |
| $0C | Word | PageRgnColor | PAT | COL1 | COL2 |
| $0E | Word | InactiveColor | 0 | BG | 0 |

OUT = Outline color
BG = Background color
FG = Foreground color
PAT = Color pattern
0 = Always zero

*ScrollOutline.* ScrollOutline is the outline color of the scroll bar, arrow boxes, and thumb.

*ArrowNorColor.* ArrowNorColor is the color of the arrow outline and background when an arrow is not being selected by the mouse.

*ArrowSelColor.* ArrowSelColor is the color of the arrow (filled) and background when the arrow is selected by the mouse. A good method of setting this and the previous color value is to reverse them: Use the foreground color for ArrowNorColor and the background color for ArrowSelColor, and vice versa.

*ArrowBackColor.* ArrowBackColor is the interior color of the arrow when it is not selected.

*ThumbNorColor.* ThumbNorColor is the color of the thumb's interior.

*ScrollReserved*. ScrollReserved is a word of 0, reserved for some secret future use.

*PageRgnColor*. PageRgnColor is the color of the page region in the scroll bar. The MSB of this word determines whether a dithered pattern is to be used. The LSB of the word contains either the solid color with which to fill the page region, or two colors to use for dithering.

If bit 8 is set, dithering takes place. The page region is filled with a checked pattern of both the colors specified in bits 7–4 and 3–0.

If bit 8 is reset to 0, the page region is filled with the solid color pattern indicated by the color specified in bits 7–4. Bits 3–0 should all be reset to 0.

Bits 15–9 of the PageRgnColor value should always be 0.

*InactiveColor*. InactiveColor is the color of the scroll bar when it has been deactivated (dimmed).

**Edit lines.** Edit lines are controls that allow a user to type a line of text into a dialog box. Edit lines are best used when the information needed by your program cannot be obtained by using a button or list of items.

Any text typed at the keyboard will appear in the edit box. Additionally, because of the LineEdit tool set, the text inside the edit line can be edited, selected with the mouse, cut, pasted, deleted, or copied to a special edit line clipboard (maintained by the Toolbox) using the standard editing keys. (See Appendix A for more on editing.)

Any key pressed will appear in the edit line. When Return is pressed, the default button of the dialog takes over and the dialog box vanishes. Because of this, if more than one edit line appears in a dialog box, the Tab key is pressed to switch between one edit line item and another. If a number of edit lines are in a single dialog box, the Tab key can be pressed repeatedly until the insert cursor is in the desired edit line.

If a default button is not defined, the Return character (an inverse question mark in the system font, or simply a blank) is displayed in the edit line just like any other character.

The first edit line defined, either by the NewDItem or GetNewDItem functions or first in a template of items for the GetNewModalDialog call, is the first edit line created and placed into the dialog. The cursor appears in the first defined edit line box. The ItemID of the edit line has nothing to do with its order.





The items listed in Table 11-1 are used to define an edit line.

**Table 11-11. Information Required to Define an Edit Line**

| Name | Size | EditLine Value |
|---|---|---|
| ItemID | Word | The EditLine's ID |
| ItemRect | Word | Upper left Y value of EditLine's box (MinY) |
| | Word | Upper left X value of EditLine's box (MinX) |
| | Word | Lower right Y value of EditLine's box (MaxY) |
| | Word | Lower right X value of EditLine's box (MaxX) |
| ItemType | Word | $0011 (17 decimal) |
| ItemDescr | Long | Pointer to string inside the EditLine, or buffer |
| ItemValue | Word | Max characters to be typed (up to 255) |
| ItemFlag | Word | Determines visibility |
| ItemColor | Pointer | Always 0 |

*ItemID*. The ItemID is a unique number used to identify the edit line. Its value is really unimportant because editing and entering text takes place automatically.

*ItemRect*. ItemRect defines a rectangle indicating the size and position of the edit line's input box in local coordinates. The length of the box (left to right) depends on the number of characters the user should be allowed to enter (and, indirectly, depends on the system font as well). The height of the box must be at least 15 pixels—anything less and text inside the edit line will not be visible.

The height of the edit line's box really depends on the size of the font used by the Dialog Box. For a smaller font, logically, a box of less than 15 pixels in height could be used. Likewise, if an exceptionally large font were being used, a height taller than 15 pixels would be required.

*ItemType*. The ItemType for an edit line is $0011, or 17 decimal.

*ItemDescr*. ItemDescr points to either a string of text that may be edited, or an empty buffer into which typed text will be placed. ItemDescr must point to something, either an empty buffer or a string of text. If ItemDescr is the address of a Pascal string of text, that text appears as selected when the Control Manager draws the edit line.

*ItemValue*. ItemValue determines how many characters are allowed inside the edit line. Only the number of characters specified

by ItemValue can be typed into the edit line, and no more. ItemValue also indirectly indicates the size of the string pointed to by ItemDescr.

*ItemFlag*. ItemFlag can be one of two values. When Itemflag is $0080, the edit line's box is invisible, but the text can still be seen. When ItemFlag is 0, EditLine's box is drawn.

The edit line control does not use a color table, so its value should be reset to a long word of 0.

## Changing Colors

Almost every control can take advantage of color. Your dialog boxes can be made colorful simply by specifying a color table pointer and filling the table with the desired values for each control. But some confusion can arise in referring to color tables as used by controls and color tables used by QuickDraw.

It should be pointed out that the color tables used when defining a control are the same as the color tables used by QuickDraw.

QuickDraw defines a color table from which certain colors are selected. For example, in the 320 mode, QuickDraw sets up a color table with 16 separate colors. Each color is defined according to the intensity of its red, green, and blue attributes. So, in a QuickDraw color table, color number 5 in that table may be set to dark green.

In the color tables used by controls, the values referred to are the values in the QuickDraw color tables. So if the current color table as used by QuickDraw has 16 values and number 5 is dark green, then when you specify a value of 5 in your control table, it takes on the color dark green. In fact, all the pixels on the super-high-resolution graphics display on the Apple IIGS work this way: They aren't fixed color values; they're simply index numbers into a color table.

Table 11-12 shows how QuickDraw assigns color values in the standard 320-mode color table. The control value and color indicate the value specified in a control's color table and the color that value represents. Use this table to determine which values in your control's color tables will take on which colors (using the standard color table in the 320 mode).





**Table 11-12. Color Values**

| QuickDraw Number | Color | Control Value | |
|---|---|---|---|
| | | Binary | Hexadecimal |
| 0 | Black | 0000 | $0 |
| 1 | Dark gray | 0001 | $1 |
| 2 | Brown | 0010 | $2 |
| 3 | Purple | 0011 | $3 |
| 4 | Blue | 0100 | $4 |
| 5 | Dark green | 0101 | $5 |
| 6 | Orange | 0110 | $6 |
| 7 | Red | 0111 | $7 |
| 8 | Beige | 1000 | $8 |
| 9 | Yellow | 1001 | $9 |
| 10 | Green | 1010 | $A |
| 11 | Light blue | 1011 | $B |
| 12 | Lilac | 1100 | $C |
| 13 | Periwinkle | 1101 | $D |
| 14 | Light gray | 1110 | $E |
| 15 | White | 1111 | $F |

A control's color table can be changed or altered to suit your personal tastes and whatever is in vogue.

## Panic Button

The following code (Programs 11-1 to 11-3) shows how a push button's size and color can be manipulated to create a very large panic button. These examples are not complete programs. The code represents a panic button subroutine (to be called at the appropriate time) that you can place into your own programs.

**Program 11-1. Panic Button in Machine Language**

```
*------------------------------*
* PANIC Button Dialog Box      *
*------------------------------*

:Equates...

DialogHeight    equ     100
DialogWidth     equ     110
ItemUsable      equ     $8000
StatText        equ     $0F
ButtonItem      equ     $0A

:Start of Routine...
```

```
Panic   pea     $0000           ;long word result space
        pea     $0000
        pushlong #DialogRecord
        _GetNewModalDialog
        jsr     ErrChk

        pullLong DialogPtr      ;get dialog pointer

;Now wait until the button is clicked

Wait    pea     $0000           ;result space
        pea     $0000           ;filter routine (long pointer)
        pea     $0000
        _ModalDialog            ;get dialog events

        pla                     ;get results
        cmp     #$1             ;was it the panic button?
        bne     Wait            ;keep waiting if not

        pushlong DialogPtr      ;we're done, close the dialog
        _CloseDialog

        rts                     ;return, done

;----Data Storage----

DialogPtr       ds 4

DialogRecord    anop
        dc      i2'(190-DialogHeight)/2'
        dc      i2'(320-DialogWidth)/2'
        dc      i2'(190-DialogHeight)/2+DialogHeight'
        dc      i2'(320-DialogWidth)/2+DialogWidth'
        dc      i2'TRUE'
        dc      i4'0'
        dc      i4'TextRecord'
        dc      i4'ButtonRecord'
        dc      i4'0'
TextRecord      anop
        dc      i2'2'
        dc      i'5,5,15,105'
        dc      i2'ItemDisable+StatText'
        dc      i4'TextString'
        dc      i2'0'
        dc      i2'0'
        dc      i4'0'

TextString      anop
        dc      i1'15'
        dc      c'It's time to...'

ButtonRecord    anop
        dc      i2'1'
        dc      i'25,5,95,105'
        dc      i2'ButtonItem'
        dc      i4'ButtonString'
        dc      i2'0'
        dc      i2'0'
        dc      i4'ColorTable
```





```
ButtonString    anop
        str     'Panic'

ColorTable      anop
        dc      i2'%0000000001010000'
        dc      i2'%0000000001111000'
        dc      i2'%0000000001111000'
        dc      i2'%0000000001111000'
        dc      i2'%0000000001111000'
        dc      i2'%0000000001111000'
```

### Program 11-2. Panic Button in C

```
/*-----------------------------*
 * PANIC Button Dialog Box     *
 *-----------------------------*/

#define DialogHeight    100
#define DialogWidth     110

ItemTemplate    TextRecord = {
                    2,
                    5, 5, 15, 105,
                    ItemDisable+statText,
                    "\pIt's time to...",
                    0, 0, NULL
};

BtnColors  ColorTable = {
                    0x0050,
                    0x00f0,
                    0x0070,
                    0x00f0,
                    0x0070
};

ItemTemplate    ButtonRecord = {
                    1,
                    25, 5, 95, 105,
                    buttonItem,
                    "\pPanic",
                    0, 0, &ColorTable
};

DialogTemplate  DialogRecord = {
                    (190 - DialogHeight) / 2,
                    (320 - DialogWidth) / 2,
                    (190 - DialogHeight) / 2 + DialogHeight,
                    (320 - DialogWidth) / 2 + DialogWidth,
                    TRUE,
                    NULL,
                    &TextRecord,
                    &ButtonRecord,
                    NULL
};

Panic()
{
    GrafPortPtr DialogPtr;
```

```
    DialogPtr = GetNewModalDialog(&DialogRecord);   ErrChk();
    while (ModalDialog(NULL) != 1);     /* Wait for PANIC button */
    CloseDialog(DialogPtr);             /* Then close the dialog */
}
```

### Program 11-3. Panic Button in Pascal

```
{ +---------------------------+
  * PANIC Button Dialog Box   *
  +---------------------------+ }

PROCEDURE Panic;

CONST   DialogHeight    = 100;
        DialogWidth     = 110;

VAR     TextRecord:     ItemTemplate;
        ButtonRecord:   ItemTemplate;
        ButtonColors:   ControlColorTbl;
        DialogRecord:   DialogTemplate;
        DialogPort:     DialogPtr;
        TextString:     String;
        ButtonString:   String;

BEGIN

    TextString := 'It's time to...';
    ButtonString := 'Panic';

    WITH TextRecord DO BEGIN
        itemID      := 2;
        SetRect     (itemRect, 5, 5, 15, 105);
        itemType    := itemDisable+StatTextItem;
        itemDescr   := @TextString;
        itemValue   := 0;
        itemFlag    := 0;
        itemColor   := nil;
    END;

    WITH ButtonColors DO BEGIN
        SimpOutline := $0050;
        SimpNorBack := $00f0;
        SimpSelBack := $007D;
        SimpNorText := $00f0;
        SimpSelText := $0070;
    END;

    WITH ButtonRecord DO BEGIN
        itemID      := 1;
        SetRect     (itemRect, 5, 25, 105, 95);
        itemType    := ButtonItem;
        itemDescr   := @ButtonString;
        itemValue   := 0;
        itemFlag    := 0;
        itemColor   := @ButtonColors;
    END;

    WITH DialogRecord DO BEGIN
        SetRect(boundsRect,
```





```
            (320 - DialogWidth) / 2,
            (190 - DialogHeight) / 2,
            (320 - DialogWidth) / 2 + DialogWidth,
            (190 - DialogHeight) / 2 + DialogHeight);
        dtVisible   := TRUE;
        dtRefCon    := 0;
        Item1Ptr    := @TextRecord;
        Item2Ptr    := @ButtonRecord;
        Terminator  := nil;
    END;

    DialogPort := GetNewModalDialog(DialogRecord);  ErrChk;
    REPEAT UNTIL ModalDialog(nil) = 1;          { Wait for PANIC button }
    CloseDialog(DialogPort);                    { Then close the dialog }
}
```

## Changing Values

This section describes how a control can be manipulated after it has been defined. Some of the functions to manipulate a control are listed under the Dialog Manager; the ones listed below are under the Control Manager.

The Control Manager must have a handle to a control before that control can be manipulated (unlike the Dialog Manager, which requires only an itemID). To get a control's handle, a call is made to the Dialog Manager's GetControlDItem function. Once the handle is obtained, the various Control Manager routines that manipulate a control can be used.

Controls can be highlighted or inactive (dimmed), visible or invisible, and selected or unselected. Make sure you know and understand these differences.

When a control is dimmed, it appears fuzzy in the dialog box. Clicking the mouse on the control will not activate it, just as selecting a dimmed menu item won't work.

A visible control is one you can see. A control can be made invisible, for example, when an option is not available, or as was demonstrated in Chapter 10, to page text.

Another attribute of a control is to be selected or unselected. This normally affects only two controls: the check box and radio button. When either of those buttons is selected, its button or box is filled, meaning whatever function it represents is active. (See the COLOR example from Chapter 10 for a demonstration.)

The following sections illustrate how the Control Manager can be used to dim, hide, or activate a control.

**Dimming controls.** The following routines will dim or highlight a control using the HiliteControl function in the Control Manager.

HiliteControl can specify whether a control is to be redrawn as normal or inactive, or whether a specific part code of the control can be individually highlighted. (The entire control is always redrawn each time HiliteControl is called.)

The parameter determining how the control is highlighted is referred to as HiliteState. It's a word-sized value, though only the least significant byte holds any meaning:

| HiliteState Value | Highlighting |
|---|---|
| 0 | Control is highlighted |
| 1–253 | Only specified parts are highlighted |
| 254 | Reserved (not used) |
| 255 | Control is dimmed |

Part codes are used to identify the individual parts of a control. In the normal operation of a DeskTop application, your program will probably never need to manipulate any individual part codes. (You'll either be dimming or highlighting the entire control.)

But, for the curious, Table 11-13 shows the part numbers defined for specific controls. Values 32–127 are available for your application's use. Any other value not listed is reserved.

**Table 11-13. Controls' Part Numbers**

| Code | | |
|---|---|---|
| Decimal | Hexadecimal | Part |
| 0 | $00 | None |
| 2 | $02 | Simple button |
| 3 | $03 | Check box |
| 4 | $04 | Radio button |
| 5 | $05 | Up arrow |
| 6 | $06 | Down arrow |
| 7 | $07 | Page up |
| 8 | $08 | Page down |
| 9 | $09 | Static text |
| 10 | $0A | Grow box |
| 11 | $0B | Edit line |
| 12 | $0C | User item |
| 13 | $0D | Long static text |
| 14 | $0E | Icon |
| 129 | $81 | Thumb |





The following code can be used to dim a control.

In machine language:

```
pushlong        #0              ;long result space
pushlong        DialogPtr       ;dialog box port pointer
pushword        ItemID          ;the control's ItemID
_GetControlDItem                ;Dialog Manager Call
pullong         ControlHandle   ;return a handle to the control
pea             255             ;dim the control
pushlong        ControlHandle
_HiliteControl
```

In C and Pascal:

```
HiliteControl(255, GetControlDItem(DialogPtr, ItemID));
```

Conversely, the following code will highlight a dimmed control (or simply redraw a highlighted control).

In machine language:

```
pushlong        #0              ;long result space
pushlong        DialogPtr       ;dialog box port pointer
pushword        ItemID          ;the control's ItemID
_GetControlDItem
pullong         ControlHandle
pea             0               ;redraw the control normal
ushlong         ControlHandle
                _HiliteControl
```

In C and Pascal:

```
HiliteControl(0, GetControlDItem(DialogPtr, ItemID));
```

**Control visibility.** The easiest way to make a control visible or invisible is by setting or resetting bit 7 of its ItemFlag. If bit 7 is reset to 0, the control is visible. If bit 7 is set to 1, the control is invisible.

The Dialog Manager functions HideDItem and ShowDItem can be used to alter the visibility of a control after it's been defined.

In machine language:

```
pushlong        DialogPtr       ;dialog box pointer
pushword        ItemID          ;ItemID of the control
_HideDItem                      ;render it invisible
jsr             ErrChk          ;test for error $150C (Item not found)
```

In C and Pascal:

```
HideDItem(DialogPtr, ItemID);
```

To make a control visible, simply replace the above HideDItem functions with ShowDItem. Note that showing an item already visible, as well as hiding an item already hidden, has no effect.

To hide a control using the Control Manager, some extra steps are required. Actually, it's recommended you use the above Dialog Manager functions. However, if you're partial to the Control Manager, you'll need to call GetControlDItem (in the Dialog Manager) to return the control's handle, then perform either the Control Manager's HideControl or ShowControl function.

In machine language:

```
pushlong        #0          ;long result space
pushlong        DialogPtr   ;dialog box port pointer
pushword        ItemID      ;the control's ItemID
_GetControlDItem
;                           ;keep the control handle on the stack
_HideControl                ;Hide it
```

In C and Pascal:

```
HideControl(GetControlDItem(DialogPtr, ItemID));
```

To show the control again, replace the HideControl functions above with ShowControl.

## Chapter Summary

The following tool set functions were referenced in this chapter.

**Function:** $0210
Name: CtlStartUp
Starts the Control Manager
Push: UserID (W); Direct Page (W)
Pull: Nothing
Errors: $1001

**Function:** $0310
Name: CtlShutDown
Shuts down the Control Manager
Push: Nothing
Pull: Nothing
Errors: None





**Function:** $0910
Name: NewControl
Creates a control
Push: Result Space (L); Window Pointer (L); Control's Rectangle (L); Title String (L); Item Flag (W); Initial Value (W); Extra Parameter 1 (W); Extra Parameter 2 (W); Definition Procedure (L); RefCon (L); Color Table (L)
Pull: Control Handle (L)
Errors: None

**Function:** $0E10
Name: HideControl
Hides a control, making it invisible
Push: Control Handle (L)
Pull: Nothing
Errors: None

**Function:** $0F10
Name: ShowControl
Shows a control, making it visible
Push: Control Handle (L)
Pull: Nothing
Errors: None

**Function:** $1110
Name: HiliteControl
Highlights or dims all or part of a control
Push: HiliteState (W); Control Handle (L)
Pull: Nothing
Errors: None

**Dialog Manager Calls**

**Function:** $0D15
Name: NewDItem
Places a control into a dialog box
Push: Dialog Pointer (L); ItemID (W); Rectangle pointer (L); ItemType (W); Item Descriptor (L); ItemValue (W); Item Flag (W); Color Table Pointer (L)
Pull: Nothing
Errors: $150A, $150B

**Function:** $1E15
Name: GetControlDItem
Returns a control handle for a dialog box item
Push: Result Space (L); Dialog Pointer (L); ItemID (W)
Pull: Control Handle (L)
Errors: $150C

**Function:** $2215
Name: HideDItem
Hides a control in a dialog box, rendering it invisible
Push: Dialog Pointer (L); ItemID (W)
Pull: Nothing
Errors: $150C

**Function:** $2315
Name: ShowDItem
Makes an item or control in a dialog box visible
Push: Dialog Pointer (L); ItemID (W)
Pull: Nothing
Errors: $150C

**Function:** $2F15
Name: SetDItemValue
Changes the value of an item, or selects an item
Push: New Item Value (W); Dialog Pointer (L); ItemID (W)
Pull: Nothing
Errors: $150C

**Function:** $3215
Name: GetNewModalDialog
Creates a modal dialog using a template
Push: Result Space (L); Template (L)
Pull: Dialog Pointer (L)
Errors: Possible Memory Manager errors

**Function:** $3315
Name: GetNewDItem
Places an item or control into a dialog box using a template
Push: Dialog Pointer (L); Template (L)
Pull: Nothing
Errors: $150A, $150B

**Memory Manager Calls**

**Function:** $0902
Name: NewHandle
Makes a block of memory available to your program
Push: Result Space (L); Block Size (L); UserID (W); Attributes (W); Address of Block (L)
Pull: Block's Handle (L)
Errors: $0201, $0204, $0207



# Chapter 12

# Interrupts

Interrupts. The very word evokes trepidation in even the most experienced programmer. Now, before you flee to the next chapter in terror, you'll find that interrupts on the IIGS are not only an essential part of the computer, but they're also a lot of fun.

The first section of this

chapter cushions the introduction to interrupts for the programmer who hasn't experienced an ordeal with them yet. It also presents the various forms of interrupts and task-switching capabilities that come as standard equipment on the Apple IIGS.

A collection of sample programs are used as the basis of study throughout the chapter, and you ought to find them exceptionally interesting, or at the very least, entertaining.

> Since interrupts involve working at the hardware level of the computer, you have to work with them in machine language. This doesn't mean that you cannot work with interrupts from C or Pascal. You can. But in order to understand the workings of interrupts, a knowledge of machine language is required. If you're a C or Pascal fan, you can take the ideas and low-level routines from the example programs in this chapter and link them with your own programs.

This chapter will concentrate on exploring the Toolbox's role in working with interrupts.

## What Are Interrupts?

An interrupt is a signal that causes the microprocessor to stop its work and momentarily switch to something else. That "something else" is called an *interrupt handler*, also known as an *interrupt service routine*. An interrupt handler takes only a split second of processor time to complete its work, and then the microprocessor returns to its previous task.

A familiar interrupt on the IIGS is the invocation of the control panel. Pressing Control–Open Apple–Escape freezes the current program and brings up a new one: the Classic Desk Accessory menu. When finished with the control panel, the program that was interrupted continues where it left off, as though nothing had ever happened. The keyboard is one part of the computer that can generate an interrupt.

In computers such as the Apple IIGS, in which many things seem to happen all at once, the ability to share slices of processor time among routines is what keeps things running smoothly. It also frees the programmer from having to watch for certain events at every turn of the program. Imagine what a pain in the flowchart it





would be if you had to keep an eye on the mouse location, move the pointer around, update the screen underneath, and so on. Since the mouse can generate interrupts when it is moved, or when its button is pressed, mouse interrupt handlers take care of all the mouse-related business behind the scenes.

Another source of interrupts is the serial port. These interrupts come into play when you have a modem connected to the computer while data is racing through the phone line. Each time a character comes through the modem and into the computer's modem port, an interrupt signal is generated. This causes a serial port interrupt handler to investigate all the commotion. When the handler discovers a character waiting at the port, it snatches the character away into a buffer, where it will be processed when the modem program is ready for it. This ensures that no characters will be lost if the computer is busy working on some other task.

Interrupts play a very important role in the operation of the Apple IIGS, especially since they are far more significant to the workings of that computer than they have been to any of its predecessors. But the correct handling of interrupts is one of the most tenuous programming tasks the budding IIGS programmer will face. Fortunately, the Apple IIGS has a few Toolbox functions that make working with interrupts easier and safer.

> Safer? Well, let's just say that if your custom interrupt handler is incorrectly written, you might find that it does a great job of reformatting your hard disk, even if you weren't writing a disk utility.
>
> Careful, precise handling of interrupts is imperative. So pay strict attention to the rest of this chapter if you haven't been scared away yet.

## Types of Apple IIGS Interrupts

In the previous section, three main sources of interrupts on the Apple IIGS were introduced: the keyboard, the mouse, and the serial port. These are considered external hardware interrupt sources

since they're activated by influences outside of the computer.

The Apple IIGS has many internal interrupts as well, mostly related to circuitry in the machine. The following is a list of some of the interrupts that can occur in an Apple IIGS:

| Type | Example Interrupt Activity |
|---|---|
| Reset | Turning on the computer |
| Reset | Control-Reset, Control–Open Apple–Reset, or Diagnostics |
| Abort | Memory fault error (from expansion RAM) |
| IRQ | Any keypress executed while the Event Manager is active |
| IRQ | Keyboard flush (Control–Open Apple–Delete) |
| IRQ | Desk Accessory menu (Control–Open Apple–Escape) |
| IRQ | Mouse movement or button press |
| IRQ | Serial port (register state changes, and so on) |
| IRQ | Firmware print spooling (buffer refresh) |
| IRQ | Video graphics controller (scan line, VBL, and so on) |
| IRQ | Ensoniq DOC (sound RAM refresh signal) |
| IRQ | Realtime clock (one second, quarter-second) |
| Software | BRK instruction encountered |
| Software | COP instruction encountered |

Interrupts come in five basic flavors:

| Interrupt | Explanation |
|---|---|
| IRQ | Maskable interrupt request |
| NMI | Nonmaskable interrupt |
| Software | Software interrupt (BRK or COP) |
| Reset | System reset interrupt |
| Abort | Memory access abort interrupt |

**Maskable interrupt request (IRQ).** A maskable IRQ interrupt is generated by a peripheral card or some other type of hardware that is physically or logically connected to the computer. A mouse, keyboard, serial port, Ensoniq DOC, clock, video graphics controller (VGC), and other such interrupt source generates IRQ interrupts. These can be masked (ignored) by the processor if the interrupt disable bit in the processor's status register is set (with the SEI instruction). Using the CLI instruction clears the disable bit, which means the processor will resume handling interrupt requests.





Just for kicks, enter the following BASIC program into Applesoft BASIC and run it.

```
10 SEI = 120 : CLI = 88 : RTS = 96
20 POKE 768, SEI
30 POKE 769, RTS
40 CALL 768
```

Now, try to bring up the Classic Desk Accessory (CDA) menu by pressing Control–Open Apple–Escape. You'll find that it refuses to pop up. This is because the 65816 processor is set to mask the interrupts that the ADB (Apple DeskTop Bus) Keyboard Micro is sending whenever the CDA menu is requested.

Change the SEI in Line 20 to CLI and rerun the program.

As soon as you press the Return key after typing RUN, the CDA menu appears. This will be discussed in detail later in the chapter.

**Nonmaskable interrupts.** Although no built-in source exists, a nonmaskable interrupt (NMI) is supported by the Apple IIGS. A nonmaskable interrupt is just like an IRQ except that (as you might guess) the processor cannot mask it out. Some Apple II peripherals, such as a screen snapshot-to-printer card or a hardware diagnostic card, can generate NMIs.

**Software interrupts.** A software interrupt can be generated by executing a BRK or COP machine language instruction. In one sense, these are nonmaskable interrupts; even if the processor's interrupt disable flag is set (SEI), a BRK instruction is still performed. BRK is used mainly for debugging purposes to insert a programmable break point in your programs. COP is intended to kick a coprocessor card—a math coprocessor, for example—into action.

**Reset interrupts.** Reset interrupts are generated mainly by pressing Control-Reset, Control–Open Apple–Reset (reboot), or Control– Open Apple–Option–Reset (diagnostics), or by turning on the computer. A reset interrupt can be simulated through software by sending a command to the Apple DeskTop Bus, or by directly calling the reset handler code in ROM ($00FA62 in emulation mode).

**Abort interrupts.** The Apple IIGS currently does not make use of an abort interrupt even though it is supported. Aborts are generated when access is made to an off-limits portion of memory, something all multi-user computers employ to keep users from poking around in other people's memory space. Should the IIGS become a true multi-user computer, this police-style interrupt would be valuable for maintaining security.

## When an Interrupt Occurs

Here's a brief rundown of what happens when the processor is interrupted (that is, as long as interrupts aren't being masked). Keep in mind that all of this happens within a few milliseconds:

- When the computer is interrupted, a program in the Apple IIGS ROM, the firmware interrupt manager, runs through a checklist of tasks to service the interrupt. It first determines which set of interrupt vectors should be used, depending on emulation mode. (These vectors are listed in Appendix B of *Mastering the Apple IIGS Toolbox*, available from COMPUTE! Books.)
- The processor speed kicks in to fast mode.
- The type of interrupt is then determined. If it's due to a BRK or COP instruction, one of the software interrupt handlers is called. If the handler is not installed, the user is sent directly to the Apple IIGS monitor.
- Machine-state information (that is, registers and flags) is saved at this point, before the serial port is tested to see whether it originated the interrupt. If it did, either *AppleTalk* or a serial port interrupt handler is called.
- Finally, if the interrupt wasn't due to a software instruction or activity at the serial port, the rest of the machine-state information is saved, and then all the other internal interrupt sources (the clock, the VGC, the mouse, and so on) in the computer are interrogated. If an internal source generated the interrupt, the interrupt manager calls the appropriate handler.
- If the interrupt wasn't from an internal source, but was from a peripheral card in one of the slots, the computer slows down to the old Apple II speed of 1 MHz, and jumps to the user interrupt vector at location $3FE in Bank $00. When ProDOS first runs, it sets this vector to point to its own internal interrupt manager. The





manager is responsible for finding some way to service the interrupt. This means that every handler associated with a peripheral card should determine whether its card generated the interrupt. The duties of such a handler are discussed later in the chapter.
- Once a handler claims the interrupt and services it, the processor restores the machine state and continues execution from the point where it was interrupted.
- However, if the interrupt is not claimed (and. as a consequence, not serviced), a fatal error occurs. If ProDOS is unable to have the interrupt serviced, it calls a fatal error handler. (In ProDOS 8 this handler would set the screen to 40 columns and display *INSERT SYSTEM DISK AND RESTART—ERR 01*). The user interrupt vector is used mainly by eight-bit data communications programs in servicing interrupts from internal modems or communications cards.

## Writing a Handler (Using Blanks)

The Toolbox provides a host of useful functions that make working with interrupts a snap. This section of the chapter will ease you into writing an interrupt handler. The first program example doesn't use interrupts, but it simulates the process of the steps required for real-life interrupt handling.

Actually, this example is quite useful (and fun). The program patches the Apple IIGS's system bell vector with a new beep. After installing this program, the computer will beep with a *fweep* sound reminiscent of a screaming banshee. No more dull, boring *bonk* sound.

The following is the plan of attack for creating the beep.

**Setup program.** First, start up just the three tool sets: Tool Locator, Miscellaneous Tools, and Memory Manager.

```
         ABSADDR     ON
         KEEP        Beep.Setup
         MCOPY       BeepMacros      :(use MACGEN to create this file)
Main     START
         phk
         plb                         ;data bank = code bank
         _TLStartUp                  ;start Tool Locator
         _MTStartUp                  ;start Misc Tools
         pha                         ;result space
         _MMStartUp                  ;start Memory Manager
         pla                         ;pull User ID
         sta         UserID
```

Next, call GetNewID to create a new User ID which will be used in allocating a new handle for the beep routine.

```
pha                     ;result space
PUSHWORD    #$A000      ;Type ID / Aux ID
_GetNewID               ;make an ID
pla
sta         CodeID
```

Then ask NewHandle to allocate a small portion of RAM with the attributes of $C018: It can be any bank or any address, does not need to be page-aligned, and cannot use special memory, cross a bank boundary, or be purged or moved at all.

```
pha                             ;result space
pha
PUSHLONG    #MBEnd-MyBeep+1     ;Size of block
PUSHWORD    CodeID              ;CodeID for this handle
PUSHWORD    #$C018              ;Fixed, locked, bolted down
PUSHLONG    #0                  ;address of the block
_NewHandle
pla                             ;get handle
plx

sta         0
stx         2
lda         [0]                 ;get long address of block
sta         BlkAddr
ldy         #2
lda         [0],y
sta         BlkAddr+2
```

Once the handle is created and its address determined, place the beep code there by using the BlockMove function. (Yes, the beep routine has to be written as relocatable code. Don't fret. The 65816 has some helpful instructions that make it possible to write relocatable code.)

```
PUSHLONG    #MyBeep             ;Source
PUSHLONG    BlkAddr             ;Destination
PUSHLONG    #MBEnd-MyBeep+1     ;Size
_BlockMove
```

Finally, SetVector is used to patch the beep vector to point to the new beep routine. This program shuts down, and you've finished.





```
            PUSHWORD    #$001B      ;Bell Vector Reference
            PUSHLONG    BlkAddr     ;New Beep Vector Address
            _SetVector
            PUSHWORD    UserID      ;shutdown everything
            _MMShutDown
            _MTShutDown
            _TLShutDown
            rtl
UserID      ds          2
CodeID      ds          2
BlkAddr     ds          4
```

The code that follows is the actual beep routine that is relocated into safe memory. Every time the IIGS is called to beep the speaker, this small routine is called.

```
Speaker     equ     $E0C030     ;speaker toggle location
MyBeep      longa   off
            longi   off
            pha                 ;preserve the registers we munge
            phy
            phx
Fweep0      ldx     #32
Fweep1      ldy     #4
Fweep3      lda     Speaker
            txa
            eso
Wait1       pha
Wait2       sbc     #1
            bne     Wait2
            pla
            sbc     #1
            bne     Wait1
            dey
            bne     Fweep3
            dex
            bne     Fweep1
            plx                 ;restore registers
            ply
            pla
            clc                 ;return with carry clear
MBEnd       rtl
            END
```

Assemble this with *APW* and run the resulting EXE file to install the new beep. (If you're hunting for a way to get the machine to beep at you so you can hear it, pull up the CDA menu and press the space bar or any other illegal key). As long as the computer is turned on, this new beep will be used in place of the old sound.

> Imagine the fun you could have with this if a digitized sound sample were played through the Ensoniq chip, rather than the all-too-common beep.
>
> If you end up liking this new beep better than the *bonk* sound the IIGS normally makes, you can make the process of patching the bell vector part of your ProDOS 16 boot sequence. Just change the file type of the EXE file to TSF ($B7) and copy it to your system disk's SYSTEM/SYSTEM.SETUP directory. It is a TSF (Temporary Startup File), because the entire program doesn't need to be kept in memory. Only the beep portion has to be retained. Every time you boot into ProDOS 16, this new sound will replace the old one, even when you're running ProDOS 8 programs.
>
> Should you wish to go back to using the standard IIGS bell sound, just move the new beep program out of the SYSTEM.SETUP directory and reboot.

This program is an excellent model for getting started on an interrupt installation and servicing program. Some important points need to be made about this program and how it relates to interrupt handlers:

- First, before writing any interrupt handler, consider the programming environment. In the case of this new beep routine, the beep code must be accessible at all times and the code must not be overwritten. That's why a special patch of RAM is allocated by NewHandle explicitly for the beep routine. Since emulation mode programs use banks $00, $01, $E0, and $E1 of the computer, the beep routine could not reside there. The beep code had to be placed outside of special memory. (See Chapter 7, which deals with memory management, for more details).





- The entire installation program is needed only once to install the beep into safe memory and set up the new bell vector. That's why NewHandle is called to allocate space only for the beep handler code. Why waste memory?
- Since NewHandle could end up placing the code anywhere in the machine, the code had to be written so that it didn't use any self-referencing addressing modes. Of course, in this example, that's not a problem. For larger applications, such a program would most likely become a relocatable load segment (more on this and other disk-related matters in Chapter 14).
- The beep routine properly maintains the environment by saving registers before changing them, and then restores them before returning. The handler should avoid modifying any other environment settings (displaying a message on the screen, changing video modes, and so on).

According to the rules, the Apple IIGS's system bell routine is always called in emulation mode with eight-bit registers and must return with the carry clear via an RTL instruction. As with an interrupt handler, there are certain steps to follow to ensure that everything is done correctly.

> Recall the sample Applesoft program from the previous section. When run, it caused the computer to ignore interrupts so you couldn't go into the CDA menu after pressing Control–Open Apple–Escape. As soon as interrupt recognition was turned on with the CLI instruction, the CDA menu popped up instantly, without your having to press Control–Open Apple–Escape again. Strange? Not at all.
>
> The reason this happened is because the interrupt of the Keyboard Micro, part of the Apple DeskTop Bus, was still pending and required servicing. The interrupt request line on the CPU was like a telephone that kept ringing until it was finally answered by the 65816 microprocessor. Once interrupt *recognition was reestablished*, the processor discovered the interrupt was pending and went out to find a way to service it. That's why the CDA menu seemed to come up all on its own. You might chalk it up to delayed reflexes.

## Interrupt Vectors

The Beep.Setup program in the last section introduces the Miscellaneous tool set's SetVector function:

**Function:** $1003
Name: SetVector
Installs an interrupt vector address
Push: Vector reference number (W); Address of routine (L)
Pull: Nothing
Errors: None
Comments: This installs the vector address, but not the interrupt service routine itself.

SetVector is used to change a multitude of system vectors and interrupt handler vectors. The vectors are identified by a unique ID number, as shown in this table:

| Reference ID | Vector Description |
|---|---|
| $0000 | Tool locator (primary) |
| $0001 | Tool locator (secondary) |
| $0002 | User's tool locator (primary) |
| $0003 | User's tool locator (secondary) |
| $0004 | Interrupt manager |
| $0005 | Coprocessor (COP) manager |
| $0006 | Abort manager |
| $0007 | System death manager |
| $0008 | *AppleTalk* interrupt handler |
| $0009 | Serial communications controller interrupt handler |
| $000A | Scan line interrupt handler |
| $000B | Sound interrupt handler |
| $000C | Vertical blanking interrupt handler |
| $000D | Mouse interrupt handler |
| $000E | Quarter-second interrupt handler |
| $000F | Keyboard interrupt handler |
| $0010 | ADB-response-byte interrupt handler |
| $0011 | ADB-SRQ interrupt handler |
| $0012 | Desk accessory manager (Control–Open Apple–Escape) |
| $0013 | Keyboard-flush-buffer handler (Open Apple–Delete) |
| $0014 | Keyboard-micro interrupt handler |
| $0015 | One-second interrupt handler |
| $0016 | External-VGC interrupt handler |
| $0017 | Other unspecified interrupt handler |
| $0018 | Cursor-update handler |
| $0019 | Increment-busy-flag routine |
| $001A | Decrement-busy-flag routine |





| | |
|---|---|
| $001B | Bell vector |
| $001C | BRK vector |
| $001D | Trace vector |
| $001E | Step vector |
| $001F–$0027 | Reserved |
| $0028 | Control-Y vector |
| $0029 | Reserved |
| $002A | ProDOS 16-MLI vector |
| $002B | Operating system vector |
| $002C | Message-pointer vector |

The actual locations in memory where the vector addresses are stored are presented in Appendix B of *Mastering the Apple IIGS Toolbox*.

SetVector's function is to install the address of a new system or interrupt handler. This is superior to the old global page scheme, where any program had access to all of the system's vectors and could destroy them accidentally. Also, using a tool to set vector addresses means that changes in vector storage locations in later ROM revisions will never be a problem.

SetVector's partner is GetVector. GetVector is used to retrieve the long address of a system/interrupt handler.

**Function:** $1103
Name: GetVector
Returns the address of an interrupt vector
Push: Result Space (L); vector reference number (W)
Pull: Vector's address (L)
Errors: None

Patching out a vector that will be used only momentarily requires the use of both of these Miscellaneous tool set functions. For example, the following routine demonstrates how you get the current vector address for the monitor's Control-Y vector, patch it out, and then restore it:

```
SetIt   pushlong    #0          ;push long result space
        pushword    #$0028      ;Vector ID = Control-Y vector
        _GetVector              ;retrieve the current address
        pullLong    OldCtrlY    ;save it for later
        pushword    #$0028      ;Vector ID = Control-Y vector
        pushlong    #NewVect    ;new Control-Y handler address
        _SetVector              ;set it
        rts
```

```
* Your program then does whatever it must do with the new
* Control-Y vector installed. Before your program quits,
* it restores the old vector address like so...

UnSetIt  pushword   #$0026
         pushlong   OldCtrlY
         _SetVector
         rts

OldCtrlY ds         4           ;long storage for old Ctrl-Y address
```

GetVector and SetVector can also be used to hook into an existing handler without actually replacing it. For example, if you wanted to have the keyboard-flush handler play a digitized sound sample of a toilet flushing, but still flush the keyboard's type-ahead buffer, you'd proceed as follows:

- Installation
  - Get the keyboard-flush handler address with GetVector.
  - Set the keyboard-flush handler vector with your own routine's address using SetVector.
- Handler operation
  - When the user presses Open Apple–Delete to flush the keyboard buffer, your handler first plays your sound sample.
  - Then it jumps to the original keyboard-flush handler address (the address obtained by the GetVector call in the installation of your handler).

### Interrupts in ProDOS 16

SetVector is one way to install an interrupt handler. You can also set one up by going through the operating system, ProDOS 16, if you prefer. This is done mainly for handlers that service interrupts from hardware installed in one of the seven peripheral slots in the IIGS.

Normally, patching into the firmware vectors with SetVector is desired because less overhead is involved since the operating system is bypassed. But the firmware vectors only support those interrupts indigenous to the circuitry in the IIGS and do not make provisions for interrupts from peripheral cards. For these, you have to go through ProDOS 16.





To install an interrupt with ProDOS 16, your program would use the ALLOC_INTERRUPT ProDOS 16 function (number $0031):

```
_ALLOC_INTERRUPT  IParms   ;Allocate the interrupt
bcs               Error    ;branch if error
```

To remove the interrupt allocation in ProDOS, the DEALLOC_INTERRUPT function is used (number $0032):

```
_DEALLOC_INTERRUPT  IParms
bcs                 Error
```

The parameter table for these calls consists of a word and a long word:

```
IParms   anop
int_num  ds      2              ;this value is returned by ProDOS
int_code dc      i4'TheHandler' ;the address of the handler
```

| Offset | Size | Description |
|---|---|---|
| +$00 | word | int_num: Interrupt handler number |
| +$02 | long | int_code: Address of interrupt handler routine |

> Actually, only the first parameter is required for DEALLOC_INTERRUPT, but in practice the same parameter block is usually referenced.

When ALLOC_INTERRUPT is used, ProDOS 16 will assign your interrupt handler a unique number which is returned in the first word, int_num. Each time you reference your handler through ProDOS, you use this number (as in the case of memory blocks with the Memory Manager).

Possible error codes returned by these calls are

| Error Code | Meaning |
|---|---|
| $07 | ProDOS is busy (it's in the middle of a command already) |
| $25 | Interrupt vector table full (there are already 16 allocated) |
| $53 | Invalid parameter (the handler's address is beyond $FFFFFF) |

If ProDOS is busy, you'll have to let it finish what it's doing and then try to allocate the interrupt again later. This is an unlikely event, unless you try to allocate another interrupt and you're already inside an interrupt handler.

Once your interrupt is allocated with ProDOS 16, you can turn on the source of the interrupt and begin handling it. When you wish to deallocate your interrupt, first turn off the interrupt source; then deallocate it.

### Environment

When an interrupt handler is called, the computer is placed into a known state, depending on the type of interrupt your handler services and how it is registered with the system. For example, an interrupt handler set up via SetVector can expect the following standard machine configuration:

Code Bank = The bank containing your handler
Data Bank = $00
Emulation = Off (Native mode)
Registers = Eight-bit widths, contents undefined, carry set
Speed = Fast

Your handler returns to the system interrupt manager via RTL.

If your handler is called from the user interrupt vector at $00/03FE, you get the same results as indicated above, except the computer will be running at 1 MHz and emulation mode will be on. Your handler returns to the system interrupt manager via RTS.

If the handler is installed through ProDOS 16, the standard configuration applies, but register widths are set to 16 bits. Your handler returns to ProDOS 16 via RTL.

If your handler modifies any registers or other environmental aspects, it must restore any changes before returning. For example, if you change register widths or their contents, you have to restore them as they were when the handler was initially called. In addition, the carry flag should be cleared before returning if your handler serviced the interrupt. If the carry is set, it indicates to the system that the interrupt was not serviced.

The typical flowchart of an interrupt handler goes something like this:

- Save all the registers and other machine-state information modified in this handler.
- Set up the environment as needed in order to service the interrupt.
- If the handler services an interrupt on a peripheral card, determine whether that card has an interrupt that needs service.
- If it doesn't, set the carry flag and return. Otherwise, service the interrupt, then clear the interrupt source. (For example, if your handler services one-second clock interrupts, it must reset that interrupt signal before returning. More on this in a later section.)
- Restore the state information saved at the beginning of the handler; then clear the carry flag and return.





Failing to restore the machine state before returning can result in some spectacularly nasty (and possibly fatal) system crashes.

### Writing a Handler

Before you can write an interrupt handler, you need to know how to turn on the source that generates interrupts. For peripheral cards in slots 1–7, you'll have to adjust the soft switches mapped to the card's slot. Directions for doing this, and other technical information about the peripheral card, should be found in its manual.

For sources built into the IIGS, the IntSource Miscellaneous tool set function is used to enable or disable interrupts for a particular source. Using it is far easier than messing with softswitches, and it keeps your hands clean, too.

**Function:** $2303
Name: IntSource
Activates or Deactivates an interrupt source
Push: Source reference number (W) (see below)
Pull: Nothing
Errors: None

| Reference Number | Description |
|---|---|
| $0000 | Enable keyboard interrupts |
| $0001 | Disable keyboard interrupts |
| $0002 | Enable vertical blanking interrupts |
| $0003 | Disable vertical blanking interrupts |
| $0004 | Enable quarter-second interrupts |
| $0005 | Disable quarter-second interrupts |
| $0006 | Enable one-second interrupts |
| $0007 | Disable one-second interrupts |
| $0008 | Reserved |
| $0009 | Reserved |
| $000A | Enable FDB data interrupts |
| $000B | Disable FDB data interrupts |
| $000C | Enable scan line interrupts |
| $000D | Disable scan line interrupts |
| $000E | Enable external VGC interrupts |
| $000F | Disable external VGC interrupts |

So, to turn on vertical blanking (VBL) interrupts, your program uses

```
pushword    #$0002   ;Enable VBL interrupts
_IntSource
```

To turn VBL interrupts off, use

```
pushword    #$0003   ;Disable VBL interrupts
_IntSource
```

Notice that all the Enable ID numbers are even, and their Disable counterparts are odd. Creative use of equates in your program can make such code self-documenting—for example:

```
Enable    gequ                0
Disable   gequ                1
VBL       gequ                2
          pushword #Enable+VBL
          _IntSource
          ;
          pushword #Disable+VBL
          _IntSource
```

> Do not attempt to turn on an interrupt source until you've installed the corresponding handler. Doing so is like starting your car while it's in first gear and the clutch is out.

The following complete program listing (Program 12-1) is an actual interrupt installation and handler. Almost as useful as changing the speaker's beep, this program will cycle through all 16 border colors around your screen. Using the one-second interrupt source on the IIGS, the border color will continue to change every second, for a little longer than a minute. It then turns off the one-second interrupts, restores the original interrupt vector, and does its best to clean up memory by unlocking its memory block for purging.





**Program 12-1. Second.ASM**

```
*------------------------------*
*         Second.ASM           *
*                              *
*  One-Second Interrupt Demo   *
*------------------------------*

          ABSADDR ON

          KEEP    Second
          MCOPY   Second.Mac      ;Create using MACGEN on this file

Main      START

          phk
          plb                     ;data bank = code bank
          _TLStartUp              ;start Tool Locator
          _MTStartUp              ;start Misc Tools
          pha                     ;result space
          _MMStartUp              ;start Memory Manager
          pla                     ;pull User ID
          sta     UserID

          pha                     ;result space
          PUSHWORD #$8000         ;Type ID / Aux ID
          _GetNewID               ;make an ID
          pla
          sta     CodeID

          pha                     ;result space
          pha
          PUSHLONG #SecEnd-OneSec ;Size of block
          PUSHWORD CodeID         ;CodeID for this handle
          PUSHWORD #$C118         ;Locked, Fixed, (purge=2)
```

```
PUSHLONG #0             ;address of the block
_NewHandle
pla                     ;get handle
plx

sta     0
stx     2
lda     [0]             ;get long address of block
sta     BlkAddr
ldy     #2
lda     [0],y
sta     BlkAddr+2

PUSHLONG #0             ;result space
PUSHWORD #$0015         ;One Second interrupt vector ID
_GetVector
PULLLONG OldVect        ;retrieve old vector address

PUSHLONG #OneSec        ;Source
PUSHLONG BlkAddr        ;Destination
PUSHLONG #SecEnd-OneSec ;Size
_BlockMove              ;move handler code
PUSHWORD #$0015         ;One Second interrupt reference #
PUSHLONG BlkAddr        ;New One Second interrupt handler address
_SetVector

PUSHWORD #$0006         ;Enable 1-sec interrupt Ref Num
_IntSource              ;turn interrupts on

PUSHWORD UserID         ;shutdown everything
_MMShutDown
_MTShutDown
_TLShutDown
_QUIT   QParms
```





```
UserID  ds      2
BlkAddr ds      4
QParms  dc      i4'0'           ;ProDOS 16 Quit Code parameters
        dc      i'$0000'

;-------------------------------;
;   Interrupt Handler Code      ;
;-------------------------------;

Border  EQU     $E0C034         ;RTC/Border color register byte
ScanInt EQU     $E0C032         ;Scanline / 1-sec interrupt source

OneSec  LONGA   OFF             ;This is the handler's entry point
        LONGI   OFF
        phb                     ;save what we end up munging
        pha
        phx
        phy

        phk
        plb                     ;data bank = code bank
        rep     #$30            ;16-bit registers
        LONGA   ON
        LONGI   ON
        per     DataSect        ;push address of data section to stack

        sep     #$20            ;accumulator = 8-bits
        LONGA   OFF
        lda     Border          ;Grab border color
        and     #$F0            ;save upper nibble (RTC bits)
        ldy     #Color-DataSect ;store to Color record in data section
        sta     (1,S),Y
        lda     Border
```

```
        inc     A               ;increment it (color is lower nibble)
        and     #$0F            ;truncate any wrapping to upper nibble
        ora     (1,S),Y         ;OR with RTC bits
        sta     Border          ;update the border
        rep     #$20            ;accumulator = 16-bits
        LONGA   ON

        ldy     #Cycle-DataSect ;get Cycle record
        lda     (1,S),Y
        dec     A               ;decrement it
        sta     (1,S),Y         ;update counter
        bne     Exit            ;if counter is not zero, exit

* Once we've cycled through the number of border changes specified,
* we turn off one-second interrupts, restore the old vector, and
* unlock this memory block to make it purgeable when needed.

        PUSHWORD #$0007         ;Disable 1-sec interrupts Ref Num
        _IntSource              ;turn 'em off first

        PUSHWORD #$0015         ;Push 1-Sec vector Ref Num
        ldy     #OldVect-DataSect+2
        lda     (1+2,S),Y       ;push high-word of old vector
        pha
        dey
        dey                     ;(index low-word)
        lda     (1+2+2,S),Y     ;push low-word of old vector
        pha
        _SetVector              ;restore old 1-sec interrupt vector

        ldy     #CodeID-DataSect
        lda     (1,S),Y         ;push code ID
        pha
        _HUnLockAll             ;unlock this block
```





```
Exit    pla                     ;pull PC relative value off stack

        sep     #$30            ;8-bit registers
        LONGA   OFF
        LONGI   OFF
        lda     #%00100000      ;clear 1-sec interrupt source
        sta     ScanInt

        ply                     ;restore registers
        plx
        pla
        plb
        clc                     ;interrupt serviced, return
        rtl

DataSect        ANOP
Color   ds      1               ;Temporary color value workspace
Cycle   dc      i'64'           ;Number of times border color changes
OldVect ds      4               ;Original 1-sec interrupt handler address
CodeID  ds      2               ;User-ID of this memory segment
SecEnd  ANOP

        END
```

Installation of the interrupt handler is similar in most respects to the Beep.Setup program listed earlier in this chapter. The only things different are

- The ID attributes for the GetNewID call do not reference a setup routine.
- The NewHandle attributes assign the memory block a purge level of 2. Even though level 3 means most purgeable, it is reserved for use by the system loader. Since the block is locked, it can't be purged until it is unlocked.
- The current vector for one-second interrupts is preserved before it's changed by the SetVector function.
- IntSource is used to turn on one-second interrupts.

Of course, the handler itself is quite different. Here is a breakdown, starting from the top and dissecting it through to the end, of what the handler does:

```
OneSec  LONGA   OFF   ;This is the handler's entry point
        LONGI   OFF
```

Since this routine is called from the firmware interrupt manager, the system will be placed into native mode with eight-bit registers. Thus, the assembler needs to be placed into the same state at the top of the routine by using the LONGA and LONGI directives.

```
phb             ;save what we end up destroying
pha
phx
phy
```

The data bank, accumulator, and X and Y registers are all changed in this routine, so they must first be saved by pushing their values onto the stack.

```
phk
plb             ;data bank = code bank
rep     #$30    ;16-bit registers
LONGA   ON
LONGI   ON
```

Next, the data bank register is set to the code bank register since this routine runs and accesses data in the same bank. It switches in 16-bit registers and tells the assembler to do likewise.

```
per   DataSect    ;push address of data section to stack
```

This is a new instruction to most 65816 programmers. PER is used to push the program counter (plus an offset) onto the stack for use in accessing portions of a relocated program. By putting the 16-bit runtime address of the program's data section on the stack, stack-relative indirect addressing can be used to access the data. This makes writing relocatable code nearly painless.

Try doing this with the venerable 6502!

```
sep     #$20                ;accumulator = 8-bits
LONGA   OFF
lda     Border              ;Grab border color
and     #$F0                ;save upper nibble (RTC bits)
ldy     #Color-DataSect     ;store to Color record in data section
sta     (1,S),Y
```





```
lda     Border
inc     A                   ;increment it (color is lower nibble)
and     #$0F                ;truncate any wrapping to upper nibble
ora     (1,S),Y             ;OR with RTC bits
sta     Border              ;update the border
rep     #$20                ;accumulator = 16-bits
LONGA   ON
```

This seemingly complicated series of instructions does one simple task: It increments the screen's border color. It starts by going into 8-bit accumulator mode and grabbing the screen's border color register (also shared by the Real Time Clock chip in the upper nibble). The RTC bits are preserved and stored in the Color data byte via stack-relative indirect addressing. The border color register is fetched once again, incremented, and then the lower nibble of the result is logically ORed with the RTC bits. Finally, the new value is stuffed back into the border color register, and the processor goes back to a 16-bit accumulator.

Most of this confusing footwork is due to the RTC bits needing to be preserved while the lower nibble of Border is incremented, all the while using stack-relative addressing.

> Any time a soft switch or $ExCxxx location is accessed, the accumulator should be set to eight bits. This is because the locations in this chunk of memory are mapped to eight-bit addresses.

```
ldy     #Cycle-DataSect     ;get Cycle record
lda     (1,S),Y
dec     A                   ;decrement it
sta     (1,S),Y             ;update counter
bne     Exit                ;If counter is not zero, exit
```

This portion of the routine decrements a counter that keeps track of the number of times the border color changes. As defined in the data section, 64 iterations will pass before the counter reaches 0. When the sixty-fourth cycle is completed, the following shutdown code is executed:

```
PUSHWORD    #$0007  ;Disable 1-sec interrupts Ref Num
_IntSource          ;turn 'em off first
```

First, the source of the one-second interrupt is shut off. This must be done before the vector is restored in case another one-second interrupt occurs in the middle of this (unlikely, but it's better to be safe than reformatted).

```
PUSHWORD    #$0018              ;Push 1-sec vector Ref Num
ldy         #OldVect-DataSect+2
lda         (1+2,S),Y           ;push high word of old vector
pha
dey
dey                             ;(index low word)
lda         (1+2+2,S),Y         ;push low word of old vector
pha
_SetVector                      ;restore old 1 sec interrupt vector
```

The vector is restored to its original setting at this point. Notice how the byte constants in the stack-relative LDAs increase by 2 each time more data is pushed onto the stack. This is because the program counter (plus data offset), initially pushed on the stack with the PER instruction, hikes up the stack each time something new is pushed, and of course, the reference must compensate for that.

```
ldy             #CodeID-DataSect
lda             (1,S),Y         ;push code ID
pha
_HUnLockAll                     ;unlock this block
```

As the last part of the shutdown sequence, the block that envelops this code is unlocked so that it can be purged whenever the Memory Manager needs to use it.

The DisposeHandle or DisposeAll functions shouldn't be used within the block being disposed. The code that follows the block could be reassigned to some other program in the computer, trashing the instructions and crashing the system.

```
Exit    pla     ;pull PC relative value off stack
```

Remember, the 16-bit address of the data section of this program is still sitting on the stack, so it must be pulled off to maintain harmony.

```
    sep     #$30        ;8-bit registers
    LONGA   OFF
    LONGI   OFF
    lda     #%00100000  ;clear 1-sec interrupt source
    sta     ScanInt
```





Once again, the computer is placed in eight-bit mode when the $ExCxxx space is being accessed. Storing $20 (%00100000) to ScanInt resets the interrupt signal for one-second interrupts. If this is not done, the processor will be beaten by this interrupt source until the signal is cleared. (For fun, you can try leaving this out just to see what happens.)

Also, recall that when the registers were saved at the top of this handler, the machine was in eight-bit mode. That means that only one byte per register is still sitting on the stack.

```
ply     ;restore registers
plx
pla
plb
clc     ;interrupt serviced, return
rtl
```

After all the registers are restored, the carry flag is cleared to indicate that the interrupt was successfully serviced. The routine returns via an RTL instruction with all registers restored and the machine still in native mode with eight-bit register widths, exactly as it was found at the beginning of this routine.

## Clearing Interrupt Sources

Part of servicing any interrupt originating from the IIGS's built-in hardware or on a peripheral card is clearing the interrupt-generating signal. This is the only way the hardware knows that someone has taken care of its interrupt. Once reset, the hardware can ready itself for new interrupts later on. If it isn't cleared, the hardware keeps the interrupt line on the microprocessor ringing nonstop.

> *Note*: Resetting an interrupt signal and disabling the source are two very different things. Disabling an interrupt source will turn it off completely, just like pulling the plug on your electric alarm clock. Resetting the interrupt signal, however, is like hitting the snooze button.

Unfortunately, there is no Toolbox function for clearing the built-in interrupt sources on the IIGS. Perhaps a future version of the Miscellaneous tool set will provide such a handy feature.

For now, your interrupt handler will have to access the hardware register area of the IIGS directly to reset interrupt signals. An example of this is the program in the previous section. It stores $20 to location $E0C032 (called SCANINT). This register contains two bits that correspond to the clearing of scan line and one-second interrupt signals. Writing a 0 to bit 6 of SCANINT resets one-second interrupts. Writing a 0 to bit 5 resets scan line interrupts. The other six bits are unused and should always be set to 0 in writing to SCANINT.

The following table identifies the interrupt reset locations in the Apple IIGS softswitch register area:

| Address | Name | Description |
|---|---|---|
| $E0C032 | SCANINT | Zero bit 6 to reset one-second interrupts; Zero bit 5 to reset scan line interrupts |
| $E0C047 | CLRVBLINT | Write to clear vertical-blanking (VBL) and quarter-second interrupts |
| $E0C048 | CLRXYINT | Write to clear mouse interrupts |

Interrupts from other sources such as serial ports can be cleared by fetching or storing data through the hardware's associated data registers.

### The Loch Ness Keyboard Interrupt

One myth about keyboard interrupts is just that: keyboard interrupts. They're a myth in and of themselves. They don't fully exist on the IIGS. The Apple IIGS keyboard really cannot generate an interrupt if, say, you press the M key. Some of the key sequences can cause interrupts, though, such as Control–Open Apple–Escape. But honest-to-goodness data interrupts from keypresses are mythical.

At the moment, keypress interrupts are simulated by some trickery built into the Apple IIGS toolbox. In essence, when IntSource is used to turn on keyboard interrupts, a special task is invoked which runs in the background every 1/60 second. This task looks at the keyboard to see whether a key was pressed, and if it was, jumps to the keyboard interrupt handler installed via SetVector. Why go about it in such a sneaky way?

Unlike most modern computers, which have keyboards that generate true interrupts from keypresses, the Apple IIGS was designed with the opinion that the extra bit of circuitry needed for





true interrupts could be sacrificed. But the IIGS's tools development team at Apple designed the Toolbox in such a way as to make a future upgrade of the hardware transparent to software. If a real interrupt-generating keyboard is available for the Apple IIGS someday, all programs that use SetVector and IntSource to establish keyboard interrupts will work just fine, and nobody will be the wiser (except you).

### In a HeartBeat

Another form of task processing on the IIGS is provided by the HeartBeat Task Manager, part of the Miscellaneous tool set. These routines allow you to add a series of tasks to perform at any number of 60Hz cycles.

> The HeartBeat Task Manager uses the vertical-blanking interrupt source, which interrupts every 1/60 second.

A HeartBeat task is a routine, usually short, that begins with a special header identifying it as a HeartBeat task. The structure of this header consists of three fields, as shown in this example:

```
TaskHdr     anop
TaskChain   dc    l4'0'       ;pointer to next task
TaskCount   ds    l'60'       ;number of 60Hz cycles before task is run
TaskSig     dc    l'$A55A'    ;special task signature
```

The TaskChain field starts out as a long value of 0. The HeartBeat manager will change this to point to the next task in the HeartBeat task queue, should another be added later.

The TaskCount word is a counter that is decremented by the HeartBeat manager every time the VBL interrupt occurs (every 1/60 second). When this counter reaches 0, your task is executed. It's up to the task to reset the counter to the appropriate number of cycles before returning. Using this method, a task can run from once every 1/60 second to once every 19 minutes.

Finally, the TaskSig word is a constant value of $A55A. If this value is not present here, an error code of $0304 (NoTaskSignature) will be returned when an attempt is made to install the task into the HeartBeat task queue.

Immediately following the task header is the code for the task itself. When the task is called, the computer is placed into native mode using 16-bit registers. The task terminates with an RTL instruction, and unlike what happens with normal interrupt handlers, absolutely nothing needs to be preserved and restored before returning. You needn't fiddle with the carry flag, and even the register widths can be left modified without causing problems. Since your task is invoked indirectly by VBL interrupts, you don't even have to reset any interrupt sources.

Indeed, this is the lazy person's way to install timed background tasks. But there are some advantages to having all the nitty-gritty details handled for you. The only disadvantage is a possible latency in execution of your task should there be a number of other tasks in the queue ahead of yours.

Installing a HeartBeat task is simple. It's done by making a call to SetHeartBeat:

**Function:** $1203
Name: SetHeartBeat
Places a task into the HeartBeat task manager queue
Push: Address of task header (L)
Pull: Nothing
Errors: $0303, Task already in queue
$0304, No task signature (or bad signature)
$0305, Damaged HeartBeat queue

As easy as using SetHeartBeat is for installing a task, the DelHeartBeat function is used to get rid of one:

**Function:** $1303
Name: DelHeartBeat
Removes a task from the HeartBeat task queue
Push: Address of task header (L)
Pull: Nothing
Errors: $0304, No task signature
$0306, Task not in queue

This chapter would be incomplete without mentioning a third HeartBeat function, ClrHeartBeat. It removes all tasks from the queue. This should never be used by your applications, though.





**Function:** $1403
Name: ClrHeartBeat
Removes all tasks from the HeartBeat task queue
Push: Nothing
Pull: Nothing
Errors: None
Comments: Don't make this call

Using the program from the previous section as a starting point, Program 12-2 installs a HeartBeat task that cycles through the border colors for about a minute. The task then removes itself gracefully.

**Program 12-2. HeartBeat.ASM**

```
*------------------------------*
*        HeartBeat.ASM         *
*                              *
*  One-Second Interrupt Demo   *
*   Using A HeartBeat Task.    *
*------------------------------*

        ABSADDR ON
        XREF    HeartBeat
        MCOPY   HB.Mac          ;create this file using MACGEN

Main    START
        phk
        plb                     ;data bank = code bank
        _TLStartUp              ;start Tool Locator
        _MTStartUp              ;start Misc Tools
        pha                     ;result space
        _MMStartUp              ;start Memory Manager
        pla                     ;pull User ID
        sta     UserID
```

```
pha                     ;result space
PUSHWORD #$FD00         ;Type ID / Aux ID
_GetNewID               ;make an ID
pla
sta     CodeID

pha                     ;result space
pha
PUSHLONG #SecEnd-OneSec ;Size of block
PUSHWORD CodeID         ;CodeID for this handle
PUSHWORD #$C118         ;Locked, Fixed, Purge=2
PUSHLONG #0             ;address of the block
_NewHandle
pla                     ;get handle
plx

sta     0
stx     2
lda     [0]             ;get long address of block
sta     BlkAddr
ldy     #2
lda     [0],y
sta     BlkAddr+2

PUSHLONG #OneSec        ;Source
PUSHLONG BlkAddr        ;Destination
PUSHLONG #SecEnd-OneSec ;Size
_BlockMove              ;move handler code

PUSHLONG BlkAddr        ;Pointer to HeartBeat task
_SetHeartBeat

PUSHWORD #$0002         ;Enable VBL interrupt Ref Num
_IntSource              ;turn interrupts on
```





```
        PUSHWORD UserID         ;shutdown everything
        _MMShutDown
        _MTShutDown
        _TLShutDown
        _QUIT   QParms

UserID  ds      2
QParms  dc      i4'0'           ;ProDOS 16 Quit Code parameters
        dc      i'$0000'
*-------------------------------*
*   Interrupt Handler Code      *
*-------------------------------*

Border  EQU     $E0C034         ;RTC/Border color register byte
Beats   EQU     60              ;HeartBeats per color change

*** Here's the task header:

OneSec  ds      4               ;task pointer storage chain
BeatCnt dc      i'Beats         ;approximately every second
        dc      i'$A55A'        ;HeartBeat task signature

*** Here's the task code:

        LONGA   OFF             ;This is the task's entry mode
        LONGI   OFF
        phk
        plb                     ;data bank = code bank
        rep     #$30            ;16-bit registers
        LONGA   ON
        LONGI   ON
        per     DataSect        ;push address of data section to stack
```

```
        sep     #$20            ;accumulator = 8-bits
        LONGA   OFF
        lda     Border          ;Grab border color
        and     #$F0            ;save upper nibble (RTC bits)
        ldy     #Color-DataSect ;store to Color record in data section
        sta     (1,S),Y
        lda     Border
        inc     A               ;increment it (color is lower nibble)
        and     #$0F            ;truncate any wrapping to upper nibble
        ora     (1,S),Y         ;OR with RTC bits
        sta     Border          ;update the border
        rep     #$20            ;accumulator = 8-bit
        LONGA   ON

        ldy     #Cycle-DataSect ;get Cycle record
        lda     (1,S),Y
        dec     A               ;decrement it
        sta     (1,S),Y         ;update counter
        bne     Exit            ;if counter is not zero, exit

* Once we've cycled through the number of border changes specified,
* we turn off VBL interrupts, remove the HeartBeat task, and
* unlock this memory block to make it purgeable when needed.

        PUSHWORD #$0003         ;Disable VBL interrupts Ref Num
        _IntSource              ;turn 'em off first

        ldy     #BlkAddr-DataSect+2     ;push address of task on stack
        lda     (1,S),Y                 ;high-word of address
        pha
        dey
        dey                     ;(index low-word)
        lda     (1+2,S),Y       ;push low-word of address
```





```
        pha
        _DelHeartBeat           ;remove this task

        ldy     #CodeID-DataSect
        lda     (1,S),Y         ;push memory block ID
        pha
        _HUnlockAll             ;unlock this block

Exit    pla                     ;pull PC relative value off stack

        per     BeatCnt         ;Update Beat counter
        ldy     #0
        lda     #Beats          ;reset HeartBeat counter for this task
        sta     (1,S),Y
        pla                     ;pull PC relative value

        rtl                     ;then return

DataSect        ANOP
Color   ds      1               ;Temporary color value workspace
Cycle   dc      i'64'           ;Number of times border color changes
CodeID  ds      2               ;User-ID of this memory segment
BlkAddr ds      4               ;Address of HeartBeat task header
SecEnd  ANOP

        END
```

The following things are new or different in the installation portion:

- No vectors are preserved.
- The task is installed with SetHeartBeat.
- VBL interrupts are turned on.

Simply installing a HeartBeat task won't make it go. The VBL interrupt source must be enabled as well.

The task portion is substantially different from the one-second interrupt handler. First, it starts with a HeartBeat task header. This task is set to execute after every 60 heartbeats, which is approximately one second. Notice that none of the processor registers are saved on the stack. This isn't needed for HeartBeat tasks.

The guts of the routine are pretty much the same: Increment the border color, and see whether 64 border changes have been made. If 64 changes have been made, the VBL interrupt source is switched off, the HeartBeat task is deleted with DelHeartBeat, and the block of memory for this task is unlocked.

Before exiting, the routine resets the task counter to 60 beats. If this isn't done, the task isn't ever called again, but remains in the queue.

Finally, the task returns to the HeartBeat manager via RTL.

### Interrupt Caveats

Here are a few important notes to keep in mind while working with interrupts:

- The example programs in this chapter use little or no error checking. The intent was to keep the program listings as simple as possible while presenting the study material. Your programs should rely heavily on error checking after each Toolbox call capable of producing errors.
- ProDOS calls and many Toolbox functions, especially those from disk-based tool sets, shouldn't be called from within an interrupt handler. Those resources might not be available at the time of the call. Instead, Apple recommends that such calls be installed into the Scheduler tool set's task queue. Information on that tool set was not available at the time of this writing.
- Switching off an interrupt source from within an interrupt handler is not a common practice. As in the HeartBeat sample program,





which can run in the background while in another application, turning off VBL interrupts can render the application useless if it depends on them.
- You shouldn't use quarter-second interrupts. These are reserved for use by *AppleTalk*.
- In general, use HeartBeat tasks for most timing-related interrupts. This is advantageous since it allows more than one such task to be present at the same time.
- Interrupt handlers are hard to debug with a runtime debugger. This is because the interrupts are occurring in realtime as you're stepping through the code.
- If, while you're programming an interrupt handler, a test run fails and causes the system to crash, it's a good idea to reboot the computer. There's no telling what has become corrupted in memory.

### Chapter Summary

Five Miscellaneous tool set functions are presented in this chapter:

- SetVector
- GetVector
- SetHeartBeat
- DelHeartBeat
- ClrHeartBeat

Their official descriptions, including stack parameters and error codes, are discussed within the text of this chapter.

# Chapter 13

# Desk Accessories

According to the Apple Human Interface Guidelines, a desk accessory is a small program that can be opened while another program is running. Good examples of desk accessories are calculators, note pads, graphic scrapbooks, alarm clocks, utilities, and games. Just about anything found on your typical



(real) desktop is considered a desk accessory.

In the Guidelines, Apple warns that desk accessories should never be too complicated. Some so-called desk accessories for the Macintosh are complete programs unto themselves: spreadsheets, word processors, and graphics programs. They go beyond the limits of desk accessories. Whether they are New or Classic, desk accessories should be quick, efficient, and helpful, short programs that make using the DeskTop interface more practical and enjoyable.

This chapter is about desk accessories. It would be silly to describe desk accessories in detail here, as if this were an introduction to the Apple IIGS. However, desk accessories are a common feature of the IIGS and Macintosh computers. They're just handy, memory-resident programs which are almost always available for use. Everything from the ever-familiar Control Panel to a modeless dialog box/alarm clock can be a desk accessory.

### Tell It to the DA

When ProDOS 16 is booted, the desk accessories stored in the SYSTEM/DESK.ACCS subdirectory are installed into memory (see Chapter 3). There can be two types of desk accessories; the advantages of each will be discussed here briefly. The first type is a Classic Desk Accessory (or CDA). This type is available at all times after ProDOS 16 is booted. Classic Desk Accessories can be chosen from the CDA menu by pressing Control–Open Apple–Escape. For example, the Control Panel (where you set your various Apple IIGS options) is merely a Classic Desk Accessory, with the exception that it's part of your ROM and isn't loaded from disk.

A New Desk Accessory (NDA) is only available to programs taking advantage of the DeskTop. NDAs are found in the Apple Menu in DeskTop applications where NDAs are specified. The FixAppleMenu ($1E05) function in the Menu Manager installs NDAs.

The key difference between CDAs and NDAs is that CDAs are always accessible via Control–Open Apple–Escape, and NDAs can only be accessed by DeskTop applications that install them. Otherwise, all desk accessories stay resident in memory until you turn off the computer, reset by pressing Control–Open Apple–Reset, or run the ROM diagnostics by pressing Control–Open Apple–Option–Reset.

It's amusing how Apple has adopted this naming convention of *New* and *Classic* desk accessories. It's suspiciously similar to the Coca-Cola Company's marketing campaign which introduced a new formula for Coke a few years ago in order to compete more successfully with Pepsi (which, as you will recall, a majority of people preferred in blind taste tests). After announcing the New Coke, they dubbed the original concoction *Coca-Cola Classic*. This is of particular interest because Apple Chairman John Sculley was lured away from PepsiCo (the people who produce Pepsi Cola) to work for Apple Computer. Just a coincidence? Apple claims it is.

Since desk accessories are memory-resident, they're usually written in machine language to make them as compact as possible. In fact, because of the structure of desk accessories, it's almost impossible to write them in a high-level language unless the compiler has special provisions for developing them.

Some high-level language compilers do make special allowances for desk accessories. The *TML Pascal* system has a special directive that places the desk accessory header information at the front of your Pascal code. This way, most of the information is handled by the compiler, and your job is simply to write the desk accessory.

Writing a desk accessory is just like writing a normal program. In fact, just about any ordinary program can be turned into a desk accessory simply by adding a bit of extra information and changing the filetype to $B8 for an NDA or $B9 for a CDA. (Note that the extra information is what's important. Simply changing a filetype does not make a desk accessory.)

The steps to creating your own desk accessory differ only in the type of desk accessory you're writing. The following sections of this chapter detail the processes of creating a Classic Desk Accessory and then a New Desk Accessory.

## Classic Desk Accessories (CDA)

Of the two types of desk accessories, the Classic Desk Accessory is simpler to program. CDAs are easy to create for two reasons. The first is that they are text-oriented. CDAs pop up on the familiar old





40- or 80-column text screen. You don't have to worry about graphics. The second reason is that they usually don't rely on DOS. Because CDAs can be used at any time, regardless of which operating system is running (ProDOS 8 or 16, DOS 3.3, Pascal, CP/M, or no DOS at all), disk-related functions should be avoided.

It should be noted that if your CDA involves disk activity, it needs to make sure the appropriate DOS is in memory. An Apple IIGS can have a CDA in memory and run another operating system. Never assume ProDOS 16 is present when, in fact, a CP/M program could be running.

If your CDA requires disk activity, it should be able to identify the current DOS environment and inform the user if it's unable to operate.

A Classic Desk Accessory begins with a special header. The header is basically text string information and pointers. For a Classic accessory, the header consists of a title and two long-word pointers:

```
MyCDA   str   "CDA Title"     ;name of DA in the CDA menu
        dc    i4'StartUp'     ;pointer to startup routine
        dc    i4'CleanUp'     ;pointer to a clean up routine
```

It may strike you as odd that this program begins with a text string. If you're sitting there wondering how the CDA can run, remember that CDA files have a special filetype that lets ProDOS know how to load and run them. For Classic Desk Accessories, a filetype of $B9 is used. Disk directories show this as a CDA filetype.

The CDA's title is a standard Pascal string (beginning with a count byte that tells the length of the string). Though it can be as many as 32 characters long, the title should be as short as possible while still being descriptive.

The long pointer to the StartUp routine is actually the address where the CDA code (program) begins. The routine at StartUp is called in full native (16-bit) mode, and it must preserve both the stack pointer (SP) and the data bank register (DBR). The routine must end with a long return (RTL). Those are the only rules to follow. The essence of the CDA resides in this routine.

The long pointer to the CleanUp code contains the address of a routine used to clean house. Whenever the DeskShutDown function is performed, this routine is called. This happens whenever

ProDOS 16 switches to ProDOS 8, or vice versa, and whenever an application makes the DeskShutDown call.

In practice, the CleanUp subroutine should be used to close files, remove interrupt handlers, and do whatever is needed to clean up any mess the CDA may have made. Like StartUp, this routine returns via an RTL. Even if there is no CleanUp routine required by your CDA, the pointer must point to an RTL instruction.

## NUMCONV.CDA

The following is a complete Classic Desk Accessory program. After assembling it, change its filetype to $B9 and copy it to your ProDOS 16 disk's SYSTEM/DESK.ACCS directory. To install it, just reboot. Then, when you need a handy hex or decimal number converter, it's only as far away as Control–Open Apple–Escape.

It might be added that this program is somewhat limited in its capabilities. Astute IIGS programmers will find ways to fix up this code or to use it as a skeleton for their own CDAs.

**Program 13-1. Number Converter CDA**

```
*------------------------------*
*       Number Converter       *
* Classic Desk Accessory Demo  *
*------------------------------*

        ABSADDR ON
        KEEP    NumConv.CDA
        MCOPY   NumConv.MAC     ;Create this file with MACGEN

NumConv START
        str     'Number Converter'      ;Name of CDA in menu
        dc      i4'StartUp'             ;Pointer to starting routine
        dc      i4'CleanUp'             ;Pointer to clean up routine

StartUp ANOP
        phb                     ;save data bank
        phk                     ;now make data bank = code bank
        plb
```





```
        _TextReset              ;initialize text I/O

        pushlong #Title         ;draw title
        _WriteCString

Loop    pushlong #Prompt        ;prompt for input
        _WriteCString

        pha                     ;result space
        pushlong #InBuf         ;pointer to input buffer
        pushword #16            ;number of characters max to read
        pushword #$8D           ;Return key (MSB set) is EOL character
        pushword #1             ;Echo input
        _ReadLine               ;get the line
        pullword Count          ;get character count
        beq     Exit            ;if equal to zero, exit

        ldx     #0              ;assume hex
        lda     InBuf           ;check for hex
        and     #$7f
        cmp     #'$
        beq     Conv            ;convert it

        ldx     #2              ;index decimal converter
Conv    jsr     (ProcTbl,X)     ;call either Hex2Dec or Dec2Hex
        bcs     Loop            ;probably string overflow

        sta     NType           ;change result prefix
        pushlong #Result        ;point to string to print
        _WriteCString
        bra     Loop            ;go back for more

Exit    plb                     ;restore bank
```

```
CleanUp ANOP                    ;No clean up needed here
        rtl                     ;return to CDA menu here
Hex2Dec pushlong #0             ;result space
        pushlong #inBuf+1       ;Point to string (skipping #)
        lda     Count
        dec     A               ;length is Count minus 1 (the #)
        pha                     ;length of string
        _Hex2Long               ;long is on the stack now
        pushlong #OutBuf        ;point to output buffer
        pushword #10            ;output string length
        pushword #0             ;unsigned
        _Long2Dec
        lda     #$A0A0          ;two spaces
        rts

Dec2Hex pushlong #0             ;result space
        pushlong #inBuf         ;point to string
        pushword Count          ;number of chars
        pushword #0             ;unsigned
        _Dec2Long               ;Long value is on stack
        pushlong #OutBuf        ;point to output buffer
        pushword #10            ;output string length
        _Long2Hex
        lda     #$A4A0          ;space / dollar sign
        rts
*-------------------------------*
*       Data Section            *
*-------------------------------*

ProcTbl dc      i'Hex2Dec'
        dc      i'Dec2Hex'

Count   ds      2
InBuf   ds      16
```





```
Result  dc      4c' .c -->'
NType   ds      2
OutBuf  dc      10c' '.i1 13,0

Title   dc      i1'12,17,15',9c' '
        dc      c'Hex Number Converter'
        dc      9c' ',i1'14,13,13'
        dc      7c' ',c'Press RETURN alone to quit',i1'13'
        dc      7c' ',c'(Start hex numbers with $)',i1'13,13,0'

Prompt: dc      i1'13',c'Numbers ',i1'0'
        END
```

If you plan to make extensive use of this desk accessory, you're advised to write a custom input routine. The ReadLine tool is adequate for getting a line of input, but doesn't allow any editing capabilities. For example, if you enter a mistake, pressing the backspace key (←) will not erase the mistake. It will insert an ASCII 8 into the input stream, causing the result of the conversion to be invalid.

## New Desk Accessories (NDA)

The formula for producing New Desk Accessories involves more ingredients than the Classic formula requires. Keep in mind the differences between the environment of the NDA and the environment of the CDA. For example, because you're in the DeskTop, it's expected that your NDA will use some form of DeskTop convention. This step alone results in a higher level of programming difficulty than that of creating the CDA.

NDAs are accessible whenever a ProDOS 16 DeskTop application is running in the super-hi-res 320 or 640 mode and the Apple [illegible] can assume that these tool sets are active and started up:

- QuickDraw
- Event Manager
- Window Manager
- Control Manager

- Menu Manager
- LineEdit
- Dialog Manager
- Scrap Manager

Of course, the Tool Locator, the Miscellaneous tool set, the Memory Manager, and other RDM-based tool sets are also available and do not require starting up or shutting down.

> No direct page space is allotted to the NDA, so it must be obtained by calling the Memory Manager's NewHandle function or by tricky use of the stack. The Magnifier program near the end of the chapter contains an example of this.

Like the Classic Desk Accessory, the NDA begins with a special header. (Also like its Classic counterpart, an NDA file can't be run directly, so it's assigned a filetype of $B8, shown as NDA in directory listings.)

The NDA header contains seven fields:

```
MyNDA   dc    i4'OpenNDA'              ;Open the NDA routine address
        dc    i4'CloseNDA'             ;Close the NDA routine address
        dc    i4'TheAction'            ;Do the NDA action routine
        dc    i4'InitNDA'              ;Init NDA StartUp or ShutDown
        dc    i'$0000'                 ;HeartBeat counter
        dc    i'$ffff'                 ;Event mask
        dc    c'--NDA Name\H',3i1'0'   ;NDA item name in Apple menu
```

The first four fields are pointers to subroutines. Each of these special routines is called by the Desk Manager as needed. They must preserve the stack and data bank registers, and end in RTL instructions. In addition, they must preserve the current GrafPort if it is swapped out. Each routine is described in more detail below.

The word value following the pointers is like a HeartBeat counter. (See Chapter 12 for details on HeartBeat tasks and interrupts.) Its value determines the number of 60Hz cycles that will pass before the NDA's Action routine is called with the Run code (more on this below). If this value is 0, the Action routine is called every pass (actually, every time the TaskMaster loop is executed in the DeskTop application). Unlike a HeartBeat task, the NDA does not need to reset this counter after each pass.





Next, a word containing an event mask is used to specify the types of events that the NDA can handle as they relate to actions concerning the NDA. The bits in this word correspond to TaskMaster Event Codes introduced in Chapter 12 of *Mastering the Apple IIGS Toolbox*.

The last field contains a text string in the format of a Menu Manager menu item line. It begins with any two characters, followed by the title of the NDA. The item line is terminated by \H and three zeros. The first two zeros are filled in by the Menu Manager with the item's ID number. The last zero is just a normal C-string terminating character.

The four special routines are described next. For real-life examples of these procedures, see Program 13-2.

The NDA Open Routine is called by the Desk Manager when it wants the NDA to create its window. In fact, the Desk Manager expects the Open routine to return a window port pointer on the stack, and provides result space for it. This is perhaps the trickiest of the four special routines, because the Open subroutine has to modify result space on the stack with the window pointer information. (You'll have a good feel for how result space is changed to meaningful values by the Toolbox after dabbling with this function.) The example NDA program below demonstrates this hair-raising procedure.

After the window is open, the Open routine should set a flag indicating that the window has been created.

The NDA Close Routine is called whenever the close box on your NDA's window is clicked, or whenever the Close menu item (ID = 255) is selected. Your NDA's Close function is used to close the window created by the Open routine. It should test the flag set by the Open routine, then close the window if it's open. Also, it should perform any other housekeeping tasks necessary to close down the NDA gracefully.

The NDA Action Routine is responsible for dispatching a host of handlers to service the events related to the NDA. When called, the Action routine will find a special code in the accumulator which corresponds to the type of action that took place. The nine Action codes are shown in Table 13-1.

**Table 13-1. Action Codes**

| Code | Type | Description |
|---|---|---|
| 1 | EventA | DeskTop event that affects the NDA has taken place. Use the X and Y registers to obtain the address of the event record to further interrogate the event. (X contains the low-order word and Y contains the high-order word of a long address.) |
| 2 | Run | It's time to run the guts of the NDA. (See the description of the HeartBeat counter field above.) |
| 3 | Cursor | If the NDA window is open, this code is passed to your Action handler each time TaskMaster is called. This is useful for changing the shape of the mouse pointer when it's moved into your NDA's window or some other area on the DeskTop. |
| 4 | Menu | A menu item has been selected. The Menu ID and Item ID are passed in the X and Y registers respectively. |
| 5 | Undo | Undo selected from the Edit Menu |
| 6 | Cut | Cut selected from the Edit Menu |
| 7 | Copy | Copy selected from the Edit Menu |
| 8 | Paste | Paste selected from the Edit Menu |
| 9 | Clear | Clear selected from the Edit Menu |

These last five codes correspond to editing functions your NDA may want to handle. If not, the NDA places a zero into the accumulator and returns. Otherwise, the NDA handles the editing action appropriately and returns with a nonzero value in the A register.

The NDA Init Routine is run whenever DeskStartUp or DeskShutDown is called by an application or by the operating system. If DeskStartUp is called, the Init routine will find that the accumulator contains a nonzero value. In this case, the NDA can do whatever it needs to do to prepare itself (usually nothing). If DeskShutDown is called, the Init routine will detect a zero value in the accumulator. It can then clean house as appropriate (for instance, it will close the NDA's window if it's still open).

## MAGNIFY.NDA

The following program is an excellent example of a New Desk Accessory. When installed and selected from the Apple menu in a DeskTop program, this NDA will bring up a small window on the screen. It magnifies 512 pixels (a 32 × 16 pixel area), at the mouse pointer's location, and draws the enlarged pixel map in its window.





It demonstrates the structure of a simple New Desk Accessory.

This will run in 640 and 320 modes, though the aspect ratio for 320 mode is a bit out of proportion (the window appears twice as wide as in 640 mode). The budding programmer will want to keep the different screen resolutions in mind when creating an NDA.

**Program 13-2. Magnifier NDA**

```
*------------------------------*
*          Magnifier           *
*    New Desk Accessory Demo   *
*------------------------------*


        ABSADDR ON
        KEEP    Magnify
        MCOPY   Magnify.MAC             ;Create using MACGEN


Magnify START
        dc      i4'OpenNDA'             ;Open the NDA
        dc      i4'CloseNDA'            ;Close the NDA
        dc      i4'TheAction'           ;Do the NDA action
        dc      i4'InitNDA'             ;Init NDA StartUp or ShutDown
        dc      i'30000'                ;Heartbeat counter: 0 = each beat
        dc      i'$ffff'                ;Event mask: $ffff = all events
        dc      c'--Magnifier\H'.3i1'0' ;NDA item name in Apple menu


*------------------------------*
*  Oper the NDA (if closed)    *
*------------------------------*


OpenNDA ANOP
        phb                     ;save data bank (+1 byte to stack)
        phk                     ;data bank = code bank
        plb
```

```
        bit     OpenFlag        ;Have we already created the window?
        bmi     Opened          ;yes -- so skip this stuff

        pha                     ;Result space
        pha
        pushlong #WindowRec
        _NewWindow              ;Create the NDA window
;                               ;Leave window port pointer on stack
        lda     1,S             ;Get window pointer (low byte) in stack
        sta     WindowPtr       ;Save in memory, and
        sta     4+1-4,S         ;Replace result space on stack

        lda     1+2,S           ;Get window pointer (high byte) in stack
        sta     WindowPtr+2     ;Save in memory, and
        sta     4-1+4+2,S       ;Replace result space on stack
;                               ;(Recall, WindowPtr is on stack)
        _SetSysWindow           ;Mark this as a system window
        dec     OpenFlag        ;(-1) Flag window as open

Opened  plb
        rtl

;-------------------------------;
;   Init NDA StartUp/ShutDown   ;
;-------------------------------;

; On entry, accumulator is zero if DeskStartUp called,
; else the A-reg is non-zero if DeskShutDown was called.
InitNDA ANOP
        tax             ;Test accumulator: DeskStartUp called?
        bne     AnRTL   ;yes -- else fall into NDA close routine...
```





```
;-------------------------------;
; Close the NDA (if not open)   ;
;-------------------------------;

CloseNDA        ANOP
        phb                     ;save data bank
        phk                     ;data bank = code bank
        plb

        bit     OpenFlag        ;is the window opened?
        bpl     Closed          ;no -- already closed

        pushlong WindowPtr
        _CloseWindow            ;Close it
        stz     OpenFlag        ;flag it closed, too

Closed  plb                     ;restore data bank
AnRTL   rtl

;-------------------------------;
;    Handle NDA Action Event    ;
;-------------------------------;

TheAction       ANOP
        phb                     ;save data bank
        phy                     ;Save X and Y (address of Event Record)
        phx

        phk                     ;set data bank = code bank
        plb
        dec     A               ;(A's range is 1..9, make it 0..8)
        asl     A               ;index procedure table
        tax
        jsr     (ProcTbl,X)     ;Handle the NDA action event
```

```
        plx                     ;Restore X...
        ply                     ;...and Y...
        plb                     ;...and DBR
        rtl

NDAUndo         ANOP            ;We don't handle any of these actions
NDACut          ANOP
NDACopy         ANOP
NDAPaste        ANOP
NDAClear        ANOP
                lda     #0      ;flag the above as not handled
NDAMenu         ANOP
NDACursor       ANOP
                rts

NDAEvent ANOP                   ;+2 bytes on stack (return address)
        phd                     ;+2 bytes to stack
        tsc
        tcd                     ;DP = SP (tricky way to get DP access)

        lda     [2+2+1]         ;Grab Event "What" code from stack
        cmp     #9              ;Event codes less than 9 supported here
        bcs     GtEq9           ;=> 9, so skip this

        asl     A               ;index into event procedure table
        tax
        jsr     (EventTbl,X)    ;call the event handler (update only)

GtEq9   pld                     ;Restore direct page
        rts
MouseDown       ANOP
MouseUp         ANOP
KeyDown         ANOP
```





```
AutoKey         ANOP
Activate        ANOP
NotUsed         rts

Update  ANOP

        pushlong WindowPtr
        _BeginUpdate            ;Set VisRgn = Update region

        jsr     NDARun          ;Magnify screen area at mouse location

        pushlong WindowPtr
        _EndUpdate              ;empty update region
        rts

;-------------------------------;
;   Main NDA "Run" Event        ;
;-------------------------------;
NDARun  ANOP
        pushlong #CurrPt        ;get current mouse location
        _GetMouse

        lda     CurrPt          ;compare current and previous points
        cmp     WorkPt
        lda     CurrPt+2        ;(could use EqualPt, but this is faster)
        sbc     WorkPt+2
        bne     Explode         ;if not equal, explode some pixels

        rts                     ;else just return

Explode movelong CurrPt,WorkPt  ;set points

        pushlong WindowPtr
```

```
        _StartDrawing           ;Draw in NDA window

        stz     MinY            ;init Y rectangle coordinates
        stz     MaxY
        moveword #16,VCount     ;Explode 16 lines down
        _HideCursor             ;turn off the pointer

VLoop   stz     MinX            ;init rectangle X params
        stz     MaxX
        inc     MaxY            ;adjust Y coord rectangle
        inc     MaxY            ;*2
        moveword CurrX.Xposn    ;reset X posn
        moveword #32,HCount     ;init horiz. counter

HLoop   pha                     ;result space
        pushlong WorkPt
        _GetPixel               ;grab the pixel at current X,Y point
        _SetSolidPenPat         ;set the pen color to it
        add     #3,MinX,MaxX
        pushlong #TheRect
        _PaintRect              ;draw the rectangle
        moveword MaxX,MinX      ;next block over
        inc     XPosn
        dec     HCount
        bne     HLoop

        moveword MaxY,MinY      ;next line down
        inc     YPosn
        dec     VCount
        bne     VLoop
        _ShowCursor             ;turn cursor back on
        movelong CurrPt.WorkPt  ;copy points for next pass comparison
        rts
```





```
;-------------------------------
;       Data Section            ;
;-------------------------------

TheRect     ANOP                ;exploded pixel rectangle
MinY        ds      2
MinX        ds      2
MaxY        ds      2
MaxX        ds      2

VCount      ds      2           ;explosion counters
HCount      ds      2

WorkPt      ANOP
YPosn       ds      2
XPosn       ds      2

CurrPt      ANOP
CurrY       ds      2           ;current mouse coordinate point

ProcTbl     dc      i'NDAEvent'     ;NDA event handler
            dc      i'NDARun'       ;actual NDA code (puts)
            dc      i'NDACursor     ;the rest of these are unused
            dc      i'NDAMenu'
            dc      i'NDAUndo'
            dc      i'NDACut'
            dc      i'NDACopy'
            dc      i'NDAPaste'
            dc      i'NDAClear'

EventTbl    dc      i'NotUsed'      ; nothing
            dc      i'MouseDown'    ; Mouse Down
            dc      i'MouseUp'      ; Mouse Up
            dc      i'KeyDown'      ; Key Down
```

```
                dc      i'NotUsed'              ; nothing
                dc      i'AutoKey'              ; Auto Key
                dc      i'Update'               ; Update * (only one handled)
                dc      i'NotUsed'              ; nothing
                dc      i'Activate'             ; Activate

OpenFlag        ds      2                       ;Boolean: NDA Window open flag
WindowPtr       ds      4                       ;GrafPortPtr: Window port pointer

wTitle          str     'Magnify'               ;Title of NDA window

WindowRec       dc      i'wRecEnd-WindowRec'    ;Size of parameter table
                dc      i'%1110000010100000'    ;Window frame bits
                dc      i4'wTitle'              ;Pointer to title
                dc      i4'0'                   ;RefCon
                dc      i'0,0,0,0'              ;Zoom rect
                dc      i4'0'                   ;Color table pointer
                dc      i'0,0'                  ;Origin (Y & X)
                dc      i'0,0'                  ;Data area (V & H)
                dc      i'0,0'                  ;Grow box max (V & H)
                dc      i'0,0'                  ;Scroll range (V & H)
                dc      i'0,0'                  ;Paging range (V & H)
                dc      i4'0'                   ;Info bar RefCon
                dc      i'0'                    ;Info bar height
                dc      i4'0'                   ;Definition procedure
                dc      i4'0'                   ;Info bar draw routine
                dc      i4'0'                   ;Content draw routine
                dc      i'40,80,72,176'         ;Position
                dc      i4'-1'                  ;Topmost plane
                dc      i4'0'                   ;Window storage pointer
wRecEnd         ANOP

                END
```





## Chapter Summary

All the tools found in the programs in this chapter are discussed in detail in other chapters of this book, as well as in *Mastering the Apple IIGS Toolbox*. None of them deal exclusively with the Desk Manager, however.

# Chapter 14

# ProDOS

Although this book is about mastering advanced programming techniques for the Toolbox on the Apple IIGS, without ProDOS such a mastery would be nearly impossible. Though they sound like two different beasts, ProDOS and the Toolbox often cross paths. For example, ProDOS is used by the



Standard File Operations tool set, Font Manager, and the Tool Locator. Those tool sets rely upon ProDOS to perform many of their functions.

### The Operating System

The *Professional Disk Operating System*, dubbed ProDOS, is little more than a handful of commands to manipulate disk drives. It isn't really an operating system in the classical sense, but it is a smart software interface between an application and a storage device.

Fully detailing the workings and command structure of ProDOS is beyond the scope of this book, so this chapter will have to serve simply as an introduction to ProDOS 16. In it, you will see how to perform a ProDOS command in machine language, C, and Pascal. Included are two lengthy sections that list the ProDOS commands and their parameters. The Standard File Operations tool set is also covered, and a sample program in machine language, C, and Pascal gives you a working example of how ProDOS is used in a real-life situation. Finally, the chapter is wrapped up with a list of ProDOS 16 error codes.

### Other Texts

If you're familiar with the way ProDOS 8 or other disk operating systems work, you'll find this chapter a useful reference. But, if you've never worked with file management, it's suggested you check out a programmer's tutorial to working with ProDOS. Some books worthy of mention are

*Apple IIGS ProDOS 16 Reference*. Apple Computer, 1987. Addison-Wesley.

*Beneath Apple ProDOS*, Worth and Lechner, 1984. Quality Software.

**A note to ProDOS 8 programmers.** You're probably familiar with ProDOS 8, the eight-bit version of ProDOS released by Apple Computer in late 1983. ProDOS 8 is the operating system that currently hosts the majority of software for the Apple II series of computers, including such popular programs as *AppleWorks*. But, since ProDOS 8 is geared toward the 64K architecture of earlier Apple IIs, it's inadequate for working with the great expanses of memory and features of the Apple IIGS. ProDOS 16 takes full advantage of the memory you have installed in your computer.

Programmers well versed in the workings of ProDOS 8 will be relieved to know that ProDOS 16 is similar to ProDOS 8 in most respects and is better in many. It's far easier to program than ProDOS 8, even though it's more sophisticated. Function calls are made in a familiar manner, the carry flag indicates that an error occurred, and so forth.

There are many new features along with the basic familiarity. Among other things, parameter tables no longer begin with a count byte. It was the intent of ProDOS 8 to verify the parameter table for a call by making sure the count byte was correct. In a way, this is useless, because the program will probably crash whether the count byte is wrong or the parameter table is referenced incorrectly.

Some of the calls have been renamed, simplified, or have slightly different parameters. Some new calls have been added to make disk operations and file management easier than ever before.

## A Call to ProDOS

Before you can use the functions in ProDOS 16, you must first boot a disk formatted and set up for ProDOS 16, such as the System Disk you received when you bought your IIGS. (See Chapter 3 for details on how a ProDOS 16 disk is set up.)

Once loaded, ProDOS 16 can be accessed from machine language by making a long jump to a subroutine at location $E100A8. For example:

```
jsl   $E100A8
```

This address is known as the ProDOS 16 Machine Language Interface (MLI) vector. Calls to the MLI vector are made in full native mode. Your program should preserve the accumulator because ProDOS 16 will store an error result in the A register after each ProDOS call is made. (More on this later.) All other registers are preserved.

A call to ProDOS is followed by two arguments:

- A command number (word)
- A pointer to a parameter list (long)

These arguments are discussed later in this chapter, but, for now, here is a typical ProDOS 16 call:

```
jsl   $E100A8      :Call the ProDOS 16 MLI
dc    i'$29'       :$29 = "Quit" command number
dc    i4'QParms'   :long pointer to parameter list
```





It might appear to be insanely dangerous to use this format for a function call. You would think that after the JSL, the program counter would return to the arguments and careen straight into bit limbo. But in fact, ProDOS 16 will adjust the program counter so that it safely returns to the instruction following the long pointer argument.

This means that you must always call the ProDOS 16 MLI via a JSL instruction, and six bytes of argument information must follow.

## Calling ProDOS from Machine Language

As shown in the previous section, calling ProDOS from machine language is done by performing a JSL to $E100A8, followed by two arguments. But the call can be simplified at the source level by using assembler macros. The *APW* Assembler's M16.PRODOS macro file contains macro definitions for every ProDOS 16 function.

Like tool calls, ProDOS 16 macros begin with an underscore, followed by the name of the ProDOS command. The argument to the macro is the address of the parameter list. As an example of using macros for doing a ProDOS call, here's the ProDOS 16 Quit function in *APW* assembler format:

```
        _QUIT  QParms      :ProDOS 16 quit function call
        . . .

;meanwhile, somewhere else in the program:

QParms  dc     i4'0'       ;longword of zero (no chaining)
        dc     i'0'        ;word of zero (no returning)
```

The _QUIT macro actually expands to the equivalent assembly language statements shown here:

```
jsl   $E100A8
dc    i'$29'
dc    i4'QParms'
```

It's obvious that macros can clean up your ProDOS 16 instructions as well as they do for Toolbox calls.

## Calling ProDOS from C and Pascal

Even though C and Pascal have their own built-in disk functions as part of their languages, your high-level programs can access ProDOS directly. The advantage is faster, more efficient programs.

The disadvantage is that your programs will be incompatible when ported to other computer environments. However, since your DeskTop applications perform tool calls and other IIGS-specific operations, it's probably safe to assume that source code compatibility has already been tossed out the window.

> A general note to C programmers: If your programs can avoid using any of the standard C library functions, including C's disk-related commands such as fopen( ), your executable program will be many times smaller.

To make a ProDOS call in C, your program should include the prodos.h header file at the top of the program:

```
#include <prodos.h>
```

This header file contains predefined symbols for error code numbers, parameter list structures, and the ProDOS function call macros.

To perform the ProDOS 16 Quit function in C, the following statement can be used:

```
QUIT( &QParms );
```

Each ProDOS function call in C follows the naming conventions of ProDOS 16: The names of the C functions are the names of the ProDOS 16 commands, and they're always in capital letters.

A ProDOS command in C requires just one argument: the address of the parameter list. The list is usually a structure containing the needed information to perform the call. Don't forget to place the ampersand (&) in front the structure name, or your program will crash.

In order to use ProDOS in *TML Pascal*, include the ProDOS16 unit symbol file in the USES portion of the program:

```
USES QDIntf,
     GSIntf,
     ProDOS16,
     MiscTools;
```

This makes all the ProDOS 16 functions available to your program. However, naming conventions for ProDOS 16 calls in *TML Pascal* are not as consistent. They all begin with "P16" and do not





include underscores. The Quit call in *TML Pascal* is

```
P16Parms.chainPath := StringPtr(0);
P16Parms.returnFlag := 0;
P16Quit( P16Parms );
```

Unfortunately, the arguments to the call are not straightforward either. All arguments to ProDOS calls in *TML Pascal* are referenced through a variant record, called P16Parms in the above example, which is of P16ParamBlk type. Before a call can be made, the fields in the parameter list record must be filled. Setting up parameter lists is discussed later.

## Checking for Errors

After each ProDOS call, and depending on which language you're using, you can check for errors:

| Language | Check for Errors |
|---|---|
| Machine language | Examine the carry flag |
| C | Check a variable |
| Pascal | Test the result of a function |

In machine language, if the carry flag is set, an error has occurred, and the accumulator will contain an error code number, as you came to expect in Toolbox calls. For example:

```
        jsl   ProDOS16MLI      ;call the ProDOS 16 MLI ($E100A8)
        dc    i'READ_BLOCK'    ;function number
        dc    i4'RBParms'      ;parameter list pointer
        bcc   NoError          ;if carry is clear, no error occurred
        jmp   HandleDiskErr    ;branch to error handler if carry set
NoError ...
```

In C, the _toolErr global variable holds a nonzero value after making a ProDOS 16 call if an error occurred. The value in _toolErr is the ProDOS 16 error code number.

```
READ_BLOCK( &RBParms );   /* Make the ProDOS 16 call */
if ( _toolErr )           /* If an error occurred... */
  HandleDiskErr( );       /* ... handle it. */
```

With *TML Pascal*, a nonzero value returned by the IOResult function indicates that an error occurred. Any positive, nonzero value is a ProDOS 16 error code number.

```
P16ReadBlock( P16Parms );     { Make the ProDOS 16 call }
IF IOResult > 0 THEN          { If an error occurred... }
  HandleDiskErr;              { ...then handle it. }
```

Error codes are provided at the end of the chapter.

## ProDOS 16 Functions

Table 14-1 is a list of the function names and numbers supported by ProDOS 16 Version 1.3, along with a short description of each command.

**Table 14-1. Functions Supported by ProDOS 16**

| | | |
|---|---|---|
| **Housekeeping Functions** | | |
| $01 | CREATE | Creates new files or directories |
| $02 | DESTROY | Destroys files or empty directories |
| $04 | CHANGE_PATH | Renames a file or directory, or moves its link |
| $05 | SET_FILE_INFO | Sets various attributes to a file |
| $06 | GET_FILE_INFO | Returns the information set by SET_FILE_INFO |
| $08 | VOLUME | Returns information about a disk volume |
| $09 | SET_PREFIX | Sets one of eight possible prefixes |
| $0A | GET_PREFIX | Gets one of the eight internal prefixes |
| $0B | CLEAR_BACKUP_BIT | Clears the backup bit on a file |
| **File Access Functions** | | |
| $10 | OPEN | Opens an existing file for reading or writing |
| $11 | NEWLINE | Specifies the newline character when reading |
| $12 | READ | Reads data from an opened file |
| $13 | WRITE | Writes data to an opened file |
| $14 | CLOSE | Closes any or all opened files |
| $15 | FLUSH | Writes any unwritten data to a file |
| $16 | SET_MARK | Changes the current position in a file |
| $17 | GET_MARK | Returns the current position in a file |
| $18 | SET_EOF | Sets the end-of-file position for a file |
| $19 | GET_EOF | Gets the end-of-file position for a file |
| $1A | SET_LEVEL | Sets the system file level for subsequent access |
| $1B | GET_LEVEL | Gets the current system file level |
| $1C | GET_DIR_ENTRY | Gets information about entries in a directory |





| | | |
|---|---|---|
| **Device Functions** | | |
| $20 | GET_DEV_NUM | Gets the device number for a device or volume |
| $21 | GET_LAST_DEV | Gets the last-accessed device number |
| $22 | READ_BLOCK | Reads a 512-byte block from a device into memory |
| $23 | WRITE_BLOCK | Writes 512 bytes from memory to a block device |
| $24 | FORMAT | Formats a device in various DOS formats |
| $25 | ERASE | Erases a formatted device |
| $2C | D_INFO | Converts a device number to its device name |
| **Environment Functions** | | |
| $27 | GET_NAME | Gets the pathname of the active application |
| $28 | GET_BOOT_VOL | Gets the volume name where PRODOS was launched |
| $29 | QUIT | Exits the current application |
| $2A | GET_VERSION | Returns the version number of ProDOS 16 |
| **Interrupt Control Functions** | | |
| $31 | ALLOC_INTERRUPT | Allocates an interrupt handler with ProDOS |
| $32 | DEALLOC_INTERRUPT | Deallocates an interrupt handler from ProDOS |

> Note that the names given here are the official names used by Apple Computer. C programmers can use these names just as they are. To use them in assembler macros, just put an underscore in front (for instance, _ERASE). For *TML Pascal* programmers, prefix each command with the letters P16 and leave out any underscores (for instance, P16GetBootVol).

Did you notice that some function numbers appear to be missing? This isn't a mistake. Apple Computer has intentionally placed "holes" in the ProDOS 16 command table for future enhancements and additions.

## Building a Parameter List

Every ProDOS call requires a parameter list in order to pass information between ProDOS and your program. In machine language, the address of the parameter list follows the command number im-

mediately after the JSL $E100A8. In C and Pascal, the argument to each ProDOS 16 function is the address of the corresponding parameter list.

Values in a parameter list consist of the types listed in Table 14-2.

**Table 14-2. Values in a Parameter List**

| Type | Size | Sample Uses |
|---|---|---|
| Constant | Word (2) | A flag, code number, bit field, reference number |
| Constant | Long (4) | File offset, block number, and so on |
| Pointer | Long (4) | Address of a pathname string or storage buffer |

> Note that unlike ProDOS 8, only word and long-word values are used in parameter lists in ProDOS 16.
>
> A long pointer to a pathname, such as a prefix, the name of a file, device, or volume, is a Pascal-style string: It begins with a count byte. All parameters that reference strings are long pointers to buffers. Never does a parameter in the list contain string data.

The layout of a sample parameter list for the OPEN ($10) function is demonstrated in Table 14-3.

**Table 14-3. Sample Parameter List**

| Size | Offset | Parameter | Description |
|---|---|---|---|
| Word | 00–01 | ref_num | Reference number |
| Long | 02–05* | pathname | Long pointer to filename string |
| Long | 06–09 | io_buffer | Address of I/O buffer |

* You must provide information in this field before making the call to ProDOS. The other fields indicate parameters returned by ProDOS. These returned values are stored in the parameter block when the call is complete. In order to make the OPEN call, all you need to do is supply the pathname pointer, the second parameter. After the call is made, ProDOS fills in the ref_num and io_buffer fields. Offsets are always shown in hexadecimal.

An example of a parameter list in use is demonstrated by this subroutine in assembly language. It makes the OPEN call and references the parameter list, OParms:

```
DoOpen      _OPEN   OParms          ;Open the file
            bcc     Okay            ;If carry is clear, the file is open
            jmp     HandleDiskErr   ;Handle the error
Okay        rts                     ;The program then does whatever
```





```
OParms      ANOP                    ;Parameter list for the OPEN function
Oref_num    ds      2               ;ref_num returned by ProDOS
Opathname   dc      I4'FileName'    ;long pointer to the filename to open
Oio_buf     ds      4               ;io_buffer address returned by ProDOS

FileName    str     '/SAMPLE/FILE'  ;Name of file to open
```

This same routine in C could be written like this:

```
OpenRec OParms = { 0,                   /* ref_num */
                   "\p/SAMPLE/FILE",    /* pathname */
                   NULL };              /* io_buf */

DoOpen( )
{
  OPEN( &OParms );
  if ( _toolErr )
    HandleDiskErr;
}
```

And, in *TML Pascal*, an equivalent procedure would be

```
PROCEDURE DoOpen;

VAR OParms: P16ParamBlk;
  FileName: String;

BEGIN
  FileName := '/SAMPLE/FILE';
  OParms.pathname2 := @FileName;
  P16Open( OParms );
  IF toResult > 0 THEN
    HandleDiskErr;
END;
```

Using the parameter table for the CLOSE function, shown in the next section, see if you can figure out how to close the file opened above by including just a single function call. It's easier than you might think.

## ProDOS 16 Parameter Tables

The following tables describe the parameter lists for every ProDOS 16 call:

> If you're programming in a high-level language, check your compiler's manuals or support files for the appropriate names of each field in a parameter list record.

**Table 14-4. Parameter Lists for Every ProDOS Call**

**$01 CREATE**

| *Size* | *Offset* | *Parameter* | *Explanation* |
|---|---|---|---|
| Long | 00–03* | pathname | Address of pathname to create |
| Word | 04–05* | access | Access bits (that is, read, write, destroy) |
| Word | 06–07* | file_type | Filetype code number ($00–$FF) |
| Long | 08–0B* | aux_type | Auxiliary filetype code ($0000–$FFFF) |
| Word | 0C–0D* | storage_type | Storage classifier ($01–$0D) |
| Word | 0E–0F* | create_date | Date when file was created (usually $0000) |
| Word | 10–11* | create_time | Time when file was created (usually $0000) |

**$02 DESTROY**

| *Size* | *Offset* | *Parameter* | *Explanation* |
|---|---|---|---|
| Long | 00–03* | pathname | Address of pathname to delete |

**$04 CHANGE_PATH**

| *Size* | *Offset* | *Parameter* | *Explanation* |
|---|---|---|---|
| Long | 00–03* | pathname | Pathname to rename or move |
| Long | 04–07* | new_pathname | New pathname or location |

**$05 SET_FILE_INFO**

| *Size* | *Offset* | *Parameter* | *Explanation* |
|---|---|---|---|
| Long | 00–03* | pathname | Address of pathname to get information on |
| Word | 04–05* | access | Access bits |
| Word | 06–07* | file_type | Filetype code number |
| Long | 08–0B* | aux_type | Auxiliary type (or total_blocks if DIR file) |
| Word | 0C–0D* | unused | |
| Word | 0E–0F* | create_date | Date when file was created |
| Word | 10–11* | create_time | Time when file was created |
| Word | 12–13* | mod_date | Date when file was modified |
| Word | 14–15* | mod_time | Time when file was modified |

**$06 GET_FILE_INFO**

| *Size* | *Offset* | *Parameter* | *Explanation* |
|---|---|---|---|
| Long | 00–03* | pathname | Address of pathname to get information on |
| Word | 04–05 | access | Access bits |
| Word | 06–07 | file_type | Filetype code number |
| Long | 08–0B | aux_type | Auxiliary type (or total_blocks if DIR file) |





| | | | |
|---|---|---|---|
| Word | 0C–0D | storage_type | Storage classifier |
| Word | 0E–0F | create_date | Date when file was created |
| Word | 10–11 | create_time | Time when file was created |
| Word | 12–13 | mod_date | Date when file was modified |
| Word | 14–15 | mod_time | Time when file was modified |
| Long | 16–19 | blocks_used | Blocks in used by this file (or volume) |

**$08 VOLUME**

| *Size* | *Offset* | *Parameter* | *Explanation* |
|---|---|---|---|
| Long | 00–03* | dev_name | Name of device to get information on |
| Long | 04–07 | vol_name | Address of buffer to store volume name |
| Long | 08–0B | total_blocks | Volume's total capacity in 512-byte blocks |
| Long | 0C–0F | free_blocks | Number of unused blocks on the volume |
| Word | 10–11 | file_sys_id | Filesystem ID (identifies disk format) |

**$09 SET_PREFIX**

| *Size* | *Offset* | *Parameter* | *Explanation* |
|---|---|---|---|
| Word | 00–01* | prefix_num | Number of the prefix to set ($0000–$0007) |
| Long | 02–05* | prefix | Address of prefix string |

**$0A GET_PREFIX**

| *Size* | *Offset* | *Parameter* | *Explanation* |
|---|---|---|---|
| Word | 00–01* | prefix_num | Number of the prefix to get ($0000–$0007) |
| Long | 02–05* | prefix | Address of returned prefix storage buffer |

**$0B CLEAR_BACKUP_BIT**

| *Size* | *Offset* | *Parameter* | *Explanation* |
|---|---|---|---|
| Long | 00–03* | pathname | Address of pathname to have its bit cleared |

**$10 OPEN**

| *Size* | *Offset* | *Parameter* | *Explanation* |
|---|---|---|---|
| Word | 00–01 | ref_num | Opened file's reference number |
| Long | 02–05* | pathname | Address of pathname to open |
| Long | 06–09 | io_buf | Address of io_buffer for this file |

**$11 NEWLINE**

| *Size* | *Offset* | *Parameter* | *Explanation* |
|---|---|---|---|
| Word | 00–01* | ref_num | Opened file's reference number |
| Word | 02–03* | enable_mask | Logical AND bitmask used against each byte |

| | | | |
|---|---|---|---|
| Word | 04–05* | newline_char | The newline character (in lower byte) |

**$12 READ**

| *Size* | *Offset* | *Parameter* | *Explanation* |
|---|---|---|---|
| Word | 00–01* | ref_num | Opened file's reference number |
| Long | 02–05* | data_buffer | Address of buffer where data is read into |
| Long | 06–09* | request_count | Number of bytes to read from file |
| Long | 0A–0D | transfer_count | Actual number of bytes read from file |

**$13 WRITE**

| *Size* | *Offset* | *Parameter* | *Explanation* |
|---|---|---|---|
| Word | 00–01* | ref_num | Opened file's reference number |
| Long | 02–05* | data_buffer | Address of data to write into the file |
| Long | 06–09* | request_count | Number of bytes to write |
| Long | 0A–0D | transfer_count | Actual number of bytes written |

**$14 CLOSE**

| *Size* | *Offset* | *Parameter* | *Explanation* |
|---|---|---|---|
| Word | 00–01* | ref_num | Opened file's reference number |

**$15 FLUSH**

| *Size* | *Offset* | *Parameter* | *Explanation* |
|---|---|---|---|
| Word | 00–01* | ref_num | Opened file's reference number |

**$16 SET_MARK**

| *Size* | *Offset* | *Parameter* | *Explanation* |
|---|---|---|---|
| Word | 00–01* | ref_num | Opened file's reference number |
| Long | 02–05* | position | How far into the file to seek |

**$17 GET_MARK**

| *Size* | *Offset* | *Parameter* | *Explanation* |
|---|---|---|---|
| Word | 00–01* | ref_num | Opened file's reference number |
| Long | 02–05 | position | Current position in file |

**$18 SET_EOF**

| *Size* | *Offset* | *Parameter* | *Explanation* |
|---|---|---|---|
| Word | 00–01* | ref_num | Opened file's reference number |
| Long | 02–05* | eof | End-of-file position (file size in bytes) |

**$19 GET_EOF**

| *Size* | *Offset* | *Parameter* | *Explanation* |
|---|---|---|---|
| Word | 00–01* | ref_num | Opened file's reference number |
| Long | 02–05 | eof | End-of-file position (file size in bytes) |





**$1A SET_LEVEL**

| *Size* | *Offset* | *Parameter* | *Explanation* |
|---|---|---|---|
| Word | 00–01* | level | New system file level for opens and closes |

**$1B GET_LEVEL**

| *Size* | *Offset* | *Parameter* | *Explanation* |
|---|---|---|---|
| Word | 00–01 | level | Current system file level |

**$1C GET_DIR_ENTRY**

| *Size* | *Offset* | *Parameter* | *Explanation* |
|---|---|---|---|
| Word | 00–01* | ref_num | Open DIR file's reference number |
| Word | 02–03* | reserved | Must be set to $0000 |
| Word | 04–05* | base | Positive or negative code for displacement |
| Word | 06–07* | displacement | Entry displacement from current entry |
| Long | 08–0B* | filename | Address of filename buffer |
| Word | 0C–0D | entry_num | Entry number |
| Word | 0E–0F | file_type | Filetype of the returned entry |
| Long | 10–13 | eof | End-of-file position (file size in bytes) |
| Long | 14–17 | blocks_used | Number of blocks in use by this entry |
| Word | 18–19 | create_date | Date file was created |
| Word | 1A–1B | create_time | Time file was created |
| Word | 1C–1D | mod_date | Date file was modified |
| Word | 1E–1F | mod_time | Time file was modified |
| Word | 20–21 | access | Access bits |
| Long | 21–24 | aux_type | Auxiliary filetype |
| Word | 25–26 | file_sys_id | Filesystem ID |

**$20 GET_DEV_NUM**

| *Size* | *Offset* | *Parameter* | *Explanation* |
|---|---|---|---|
| Long | 00–03* | dev_name | Address of device name string |
| Word | 04–05 | dev_num | The device number of the named device |

**$21 GET_LAST_DEV**

| *Size* | *Offset* | *Parameter* | *Explanation* |
|---|---|---|---|
| Word | 00–01 | dev_num | Last accessed device number |

**$22 READ_BLOCK**

| *Size* | *Offset* | *Parameter* | *Explanation* |
|---|---|---|---|
| Word | 00–01* | dev_num | Device number to read from |
| Long | 02–05* | data_buffer | Address of 512-byte data buffer |
| Long | 06–09* | block_num | Block number to read |

**$23 WRITE_BLOCK**

| *Size* | *Offset* | *Parameter* | *Explanation* |
|---|---|---|---|
| Word | 00–01* | dev_num | Device number to write to |
| Long | 02–05* | data_buffer | Address of 512-byte data buffer |
| Long | 06–09* | block_num | Block number to write |

**$24 FORMAT**

| *Size* | *Offset* | *Parameter* | *Explanation* |
|---|---|---|---|
| Long | 00–03* | dev_name | Address of device name to format |
| Long | 04–07* | vol_name | Address of the device's new volume name |
| Word | 08–09* | file_sys_id | Filesystem ID code |

**$25 ERASE**

| *Size* | *Offset* | *Parameter* | *Explanation* |
|---|---|---|---|
| Long | 00–03* | dev_name | Address of device name to erase |
| Long | 04–07* | vol_name | Address of the device's new volume name |
| Word | 08–09* | file_sys_id | Filesystem ID code |

**$27 GET_NAME**

| *Size* | *Offset* | *Parameter* | *Explanation* |
|---|---|---|---|
| Long | 00–03* | data_buffer | Address of application's pathname buffer |

**$28 GET_BOOT_VOL**

| *Size* | *Offset* | *Parameter* | *Explanation* |
|---|---|---|---|
| Long | 00–03* | data_buffer | Address of boot volume's name buffer |

**$29 QUIT**

| *Size* | *Offset* | *Parameter* | *Explanation* |
|---|---|---|---|
| Long | 00–03* | pathname | Address of pathname to quit to |
| Word | 04–05* | flags | Return and Restart flags in bits 15 & 14 |

**$2A GET_VERSION**

| *Size* | *Offset* | *Parameter* | *Explanation* |
|---|---|---|---|
| Word | 00–01 | version | Major and minor release versions of ProDOS |

**$2C D_INFO**

| *Size* | *Offset* | *Parameter* | *Explanation* |
|---|---|---|---|
| Word | 00–01* | dev_num | Device number to convert |
| Long | 02–05* | dev_name | Address of 32-byte device name buffer |





**$31 ALLOC_INTERRUPT**

| *Size* | *Offset* | *Parameter* | *Explanation* |
|---|---|---|---|
| Word | 00–01 | int_num | Interrupt reference number |
| Long | 02–05* | int_code | Address of interrupt handling routine |

**$32 DEALLOC_INTERRUPT**

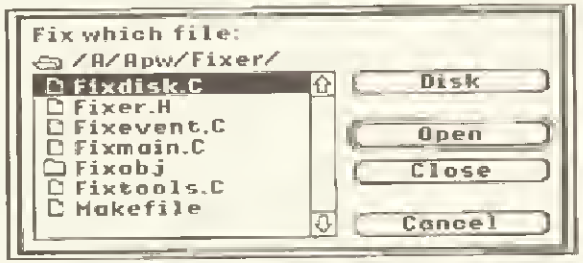
| *Size* | *Offset* | *Parameter* | *Explanation* |
|---|---|---|---|
| Word | 00–01* | int_num | Interrupt reference number |

* You must provide information in this field before making the call to ProDOS. The other fields indicate parameters returned by ProDOS. These returned values are stored in the parameter block when the call is complete. In order to make the OPEN call, all you need to do is supply the pathname pointer, the second parameter. After the call is made, ProDOS will fill in the ref_num and io_buffer fields. Offsets are always shown in hexadecimal.

## Standard File Operations

Tool set 23 ($17), the Standard File Operations tool set, contains a handful of functions that make file selection easier for both the programmer and the user of the program. These tools work in the super-hi-res graphics displays in 320 or 640 modes and present the user with a dialog box containing a list of selectable filenames.

**Figure 14-1. A Standard File Operations Dialog Box**




In addition to the standard housekeeping calls (StartUp, Status, and so on) the Standard File Operations tool set provides the functions shown in Table 14-5.

**Table 14-5. Functions Provided by Standard File Operations Tool Set**

| ID | Function | Description |
|---|---|---|
| $0917 | SFGetFile | Allows the user to select a file to open |
| $0A17 | SFPutFile | Lets the user choose a file to be saved |
| $0B17 | SFPGetFile | Same as SFGetFile, except uses a custom dialog |
| $0C17 | SFPPutFile | Same as SFPutFile, except uses a custom dialog |
| $0D17 | SFAllCaps | Chooses uppercase or mixed case filename displays |

Note that these are toolbox calls, not ProDOS 16 commands.

## SFGetFile

Use SFGetFile when your program prompts the user to select a file to open. Some examples follow.

In machine language:

```
pushword    #WhereX      ;left coordinate of dialog box
pushword    #WhereY      ;top coordinate of dialog box
pushlong    #Prompt      ;address of prompt string
pushlong    #FilterProc  ;address of filter procedure
pushlong    #TypeList    ;address of valid filetypes list
pushlong    #Reply       ;address of reply record
_SFGetFile
```

In C:

```
SFGetFile( whereX, whereY, "\pPrompt", &filterProc, &typeList, &reply );
```

In Pascal:

```
SFGetFile( WhereX, WhereY, 'Prompt', @FilterProc, @TypeList, Reply );
```

The WhereX and WhereY values specify the position on the screen where the dialog box will be placed.

The Prompt string is a Pascal string which is displayed at the top of the dialog box. This should indicate to the user the purpose of the dialog box by giving a one-line instruction, such as *Select a file to open:*.

Your program may not want the user to be able to select certain files. So you can write your own filter procedure to determine how files are to be displayed in the list. If the address of FilterProc is 0, no filter procedure is called. Otherwise, your filter routine is called for every entry to be placed into the scrolling filename list.





FilterProc is invoked in the following manner by the Standard File tool set:

```
pushword    #0           ;result space that you will fill in
pushlong    #CurrentEntry ;the address of a directory entry
jsl         YourFilterProc ;then your filtering routine is called
pullword    ResultCode   ;pull result code
```

As shown, your filter routine must access the two arguments on the stack in order to specify how the current directory entry should be placed in the list. Note that when your filter routine is called, the stack will contain a long return address, followed by a long pointer to a directory entry structure and a word of result space.

The result that your filter routine returns is one of three values:

| Value | Meaning |
|---|---|
| 0 | Do not place the entry into the dialog window |
| 1 | Place it in the window, but make it dimmed and not selectable |
| 2 | Place it in the window and allow it to be selected |

Since the filter procedure must access each file's directory record, you need to know the structure of this 39-byte buffer. This structure is shown in Table 14-6.

**Table 14-6. Structure of Directory Record**

| Offset | Field | Directory Entry Description |
|---|---|---|
| 00 | storage_type | Storage classifier (upper nibble) |
| | name_length | Filename length (lower nibble) |
| 01–0F | file_name | String of characters for filename |
| 10 | file_type | Filetype code ($00–$FF) |
| 11–12 | key_pointer | Pointer to index block |
| 13–14 | blocks_used | Number of blocks in use by this entry |
| 15–17 | eof | End-of-file position (file's size in bytes) |
| 18–19 | create_date | Date file was created |
| 1A–1B | create_time | Time file was created |
| 1C | version | Version of ProDOS that created this file |
| 1D | min_version | Oldest version of ProDOS that can use this file |
| 1E | access | Access bits |
| 1F–20 | aux_type | Auxiliary filetype |
| 21–22 | mod_date | Date file was last modified |
| 23–24 | mod_time | Time file was last modified |
| 25–26 | header_pointer | Block number of this file's parent directory |

A few fields in this record contain byte values, so you might have to put the processor in eight-bit mode for some operations.

The most straightforward way to filter out a directory entry is done as shown in the following routine. It checks the filetype of the current entry to see how the entry should be displayed:

```
DirEntry    equ     $FC             ;direct page storage for a long pointer
MyFilter    pullong Return          ;Pull RTL address off stack
            pullong DirEntry        ;set up a long pointer to the entry record
            pla                     ;unload result space from stack for now
            ldy     #$10            ;index into filetype field of entry record
            lda     [DirEntry],Y    ;grab the filetype byte (and next byte)
            and     #$FF            ;make only the filetype byte significant
            ldx     #$00            ;X=0: do not display (assume BAD)
            cmp     #$01            ;is it a BAD block file?
            beq     Done            ;yes
            inx                     ;X=1: display as dimmed (not selectable)
            cmp     #$0D            ;is it a DIR file?
            beq     Done            ;yes
            inx                     ;X=2: display and make selectable
Done        phx                     ;push filter code on stack
            pushlong Return         ;put return address back on stack
            rtl                     ;and return to it
Return      ds      4               ;Storage for return address
```

Once control returns to SFGetFile, it pulls the filter code off the stack and knows how to handle the entry.

Another way to filter entries is to provide a list of acceptable filetypes by pointing to a TypeList table. Only the file entries which have types listed in the table will be placed into the dialog box. A TypeList begins with a count byte (not a word) followed by a string of byte values indicating valid filetypes. For example:

```
TypeList  dc  i1'4','04,0B,1A,B0'  ;Four document types
```

If you specify a null address for a TypeList, or the list begins with a count byte of 0, this added filtering method is ignored. But, if you specify both a FilterProc and a TypeList, your filter procedure will be called only for the entries that satisfy the file types in the TypeList.

The final argument to the SFGetFile tool call is the address of a Reply record in the format shown in Table 4-6.





**Table 4-6. Format of the Reply Record**

| Offset | Field Name | Reply Record Description |
|---|---|---|
| $00–01 | good | True if Open clicked; false if Cancel clicked |
| $02–03 | file_type | Filetype code of the file selected |
| $04–05 | aux_file_type | Auxiliary filetype code |
| $06–15 | filename | Name selected from name list (16 bytes) |
| $16–96 | full_pathname | Full pathname to file (129 bytes) |

This record is filled in with values by SFGetFile whenever the user clicks the Open or Cancel buttons.

Your program will know whether it should continue with file operations by examining the good field of the Reply record. If it contains a false (0) value, the program knows that the user clicked Cancel. Any nonzero value means the Open button was clicked.

SFGetFile also returns the filetype and aux_file_type codes for the file selected. This information might be useful to your program.

The filename and full_pathname fields are Pascal-style strings. The 15-character filename is the name of the file selected as it was shown in the scrollable list (mixed case and all). The full_pathname is a fully qualified pathname to the file selected.

After a file is chosen, the current ProDOS prefix is set to the subdirectory (or, the folder) that contains the selected file.

## SFPutFile

Use the SFPutFile function to allow the user to select a file when saving information to disk. If the user selects a file that already exists, SFPutFile will bring up a second dialog box on its own to ask the user whether it's okay to overwrite the existing file.

In machine language:

```
pushword  #WhereX     ;left edge of dialog
pushword  #WhereY     ;top edge
pushlong  #Prompt     ;address of prompt string
pushlong  #OrigName   ;address of original filename
pushword  #MaxLen     ;maximum number of characters in name
pushlong  #Reply      ;address of reply record
```

In C:

```
SFPutFile( whereX, whereY, "\pPrompt", &origName, maxLen, &reply );
```

In Pascal:

```
SFPutFile( WhereX, WhereY, 'Prompt', @OrigName, MaxLen, Reply );
```

The WhereX and WhereY values specify the position on the screen where the dialog box will be placed.

The Prompt string is a Pascal string and should provide a message, such as *Save document to:*, giving the user an idea of the operation at hand.

OrigName is the address of a Pascal string to be placed into the EditLine item in the SFPutFile dialog. OrigName normally points to the filename returned by SFGetFile when the file was first opened.

MaxLen indicates the number of characters that can be entered in the EditLine item. This is usually 15 since the current implementation of ProDOS limits filenames to 15 characters.

The last argument is the address of the Reply record as described in the SFGetFile section. Both SFGetFile and SFPutFile use the same Reply record format.

### SFAllCaps

Normally filenames are shown in mixed case in the scrolling list of names in a Standard File dialog box, but if you prefer all uppercase, use the SFAllCaps function with a Boolean value of true (any nonzero value). A false value (0) indicates mixed case.

In machine language:

```
pushword    #1   ;true: show names in all uppercase
_SFAllCaps
```

In C and Pascal:

```
SFAllCaps( TRUE );
```

### A Nonredundant Example

Because ProDOS calls occupy a relatively small portion of the sample program for this chapter, things will be handled a bit differently. Rather than providing three huge programs in machine language, C, and Pascal, only one program is presented in its entirety. C was chosen for the job because it is midway between the low-level control of machine language and the high-level ease of Pascal. The section containing the ProDOS calls is provided in both machine language and Pascal, however.

The program listed below, CRC.C, is a 320-mode desktop program that calculates a cyclic redundancy checksum on the contents of a disk file. Unlike most of the other programs in this book,





CRC.C uses no pull-down menus. Instead, the program is centered around the Standard File Operation's SFGetFile dialog box on the desktop. The user selects a file and clicks the Open button to begin the CRC calculation. To quit the program, the user simply clicks on the Cancel button. Putting a pull-down menu into a program like this would introduce an unnecessary step, so menus are left out.

> What in the world is a CRC? A CRC is a calculation on a piece of data that results in a unique 16-bit value. It's used mainly in data communications protocols to ensure the correct transfer of a file over less-than-pristine telephone connections. For everyday purposes, it can be used to quickly compare two files that are supposed to be identical to see if they are different.

### CRC.C

This program demonstrates how to use the SFGetFile function to allow the user to select a file from disk. It will open the selected file, read through it, trap the famous "end-of-file" error, and close the file; a typical file-handling procedure. Note also how this program can easily be changed to run in 640 mode just by modifying two definitions near the top of the program.

**Program 14-1. CRC.C**

```
/*-------------------------------------------*
 *                  CRC.C                    *
 *   Cyclic Redundancy Checksum Calculator   *
 *          Written by Morgan Davis          *
 *-------------------------------------------*/


#include <types.h>
#include <locator.h>
#include <memory.h>
#include <misctool.h>
#include <quickdraw.h>
#include <qdaux.h>
```

```
#include <event.h>
#include <window.h>
#include <lineedit.h>
#include <control.h>
#include <dialog.h>
#include <stdfile.h>
#include <intmath.h>
#include <prodos.h>

#define Mode        320                 /* Screen Mode (320 or 640) */
#define MasterSCB   mode320             /* MaxWidth mode (320 or 640) */

#define Center      ((Mode - 1) / 2)    /* Center pixel column */
#define BoxWidth    240                 /* Dialog box size & location */
#define BoxHeight   70
#define BoxX        (Center - (BoxWidth / 2))
#define BoxY        40

#define BufferSize  2048        /* Size of file input buffer */

GrafPortPtr DialogPort;         /* Dialog port */
WmTaskRec   EventRec;           /* Event Record Structure */
SFReplyRec  Reply;              /* Standard File Record Structure */
OpenRec     OParms;             /* Open File parameter list */
FileIORec   RParms;             /* Read File parameter list */
QuitRec     QParms = { 0L, 0 }; /* Quit parameter list */

Word        UserID,             /* Our User ID */
            MemID,              /* Memory Management ID */
            CRC;                /* The CRC */
```





```
Word    ToolStt[] = { 6,
            14, 0,  /*          Window Manager */
            16, 0,  /* 0x100    Control Manager */
            18, 0,  /*          QuickDraw II Aux */
            20, 0,  /* 0x100    LineEdit */
            21, 0,  /*          Dialog Manager */
            23, 0   /* 0x100    Standard File */
};                  /* 0x300    QuickDraw II */
                    /* 0x100    Event Manager */
                    /* =====    total direct page space is ... */
#define DPageSize       0x700L

char        *DPBase;            /* Direct Page base pointer */

ItemTemplate OKItem = { 04,
                        BoxHeight-22, BoxWidth-68, 0, 0,
                        buttonItem,
                        "\p OK ",
                        0, 0, NULL
};

DialogTemplate CRCBox = {
                        BoxY, BoxX, BoxY+BoxHeight, BoxX+BoxWidth,
                        1,
                        NULL,
                        &OKItem,
                        NULL
};
```

```
/*-----------------------------*
 *    Handle Toolbox Errors    *
 *-----------------------------*/

ErrChk()                        /* Check for error, die if so */
{
    if (_toolErr) SysFailMgr(_toolErr, NULL);
}

/*-----------------------------*
 *  Manage Direct Page Buffers *
 *-----------------------------*/

char    *GetDP(bytes)
Word    bytes;
{
    char *OldDP = DPBase;
    DPBase += bytes;            /* Update base level pointer */
    return (OldDP);             /* Return old DPBase pointer */
}

/*-----------------------------*
 *       Start Up Tools        *
 *-----------------------------*/

StartUpTools()
{
    Word    GetDP();            /* Force words from GetDP */

    TLStartUp();                                        ErrChk();
    MemID = (UserID = MMStartUp() | 256;                ErrChk();
```





```
    MTStartUp();                                        ErrChk();
    DPBase = *(NewHandle(DPageSize, MemID, 0xc005, NULL));  ErrChk();
    QDStartUp(GetDP(0x300), MasterSCB, 0, UserID);      ErrChk();
    EMStartUp(GetDP(0x100), 0x14, 0, Mode, 0, 200, UserID);  ErrChk();

    SetForeColor(9);
    SetBackColor(0);
    MoveTo(20,20);
    DrawCString("One moment...");
    InitCursor();

    LoadTools(ToolList);                ErrChk();
    QDAuxStartUp();                     ErrChk();
    WindStartUp(UserID);                ErrChk();
    CtlStartUp(UserID, GetDP(0x100));   ErrChk();
    LEStartUp(UserID, GetDP(0x100));    ErrChk();
    DialogStartUp(UserID);              ErrChk();
    SFStartUp(UserID, GetDP(0x100));    ErrChk();

    Desktop(5, 0x40000030);
}

/*-----------------------------*
 *  Calculate CRC on a Buffer  *
 *-----------------------------*/

/* A CRC is the result of a mathematical operation based on the
 * coefficients of a polynomial when multiplied by X^16 then divided by
 * the generator polynomial (X^16 + X^12 + X^5 + 1) using modulo two
 * arithmatic.  That's okay, I don't understand it either.
 */
```

```
CalcCRC (ptr, count)
char    *ptr;           /* Pointer to start of data buffer */
Word    count;          /* Number of bytes to scan through */
{
    Word x;

    do {
        CRC  = *ptr++ << 8;             /* XOR hi-byte of CRC w/ data */
        for (x = 8; x; --x)             /* Then, for 8 bit shifts...  */
            if (CRC & 0x8000)           /* Test hi order bit of CRC   */
                CRC = CRC << 1  0x1021; /* if set,shift & XOR w/$1021 */
            else
                CRC <<= 1;              /* Else, just shift left once.*/
    } while (--count);                  /* Do this for all bytes      */
}

/*-----------------------------*
 *      Get CRC on the File    *
 *-----------------------------*/

GetCRC (pathname)
char    *pathname;      /* Pointer to full pathname */
{
    Word    Error;
    Boolean EOF = FALSE;
    char    Buffer[BufferSize];

    OParms.openPathname = pathname;
```





```
    OPEN (&OParms);                         /* Open the file */
    Error = _toolErr;
    if (!Error) {                           /* If no error ... */
        RParms.fileRefNum = OParms.openRefNum;
        RParms.dataBuffer = Buffer;
        RParms.requestCount = BufferSize;
        do {
            READ (&RParms);                 /* ...read some data */
            Error = _toolErr;
            if (Error) {                    /* If error... */
                EOF = TRUE;                 /* flag EOF */
                if (Error == eofEncountered)    /* EOF isn't fatal... */
                    Error = 0;              /* ...so zero error */
            } else
                CalcCRC(Buffer, RParms.transferCount);
        } while (!EOF);
    }
    CLOSE (&OParms);                    /* Close the file */
    return (Error);                     /* And return error code */
}

/*-----------------------------*
 *   Show CRC in Modal Dialog  *
 *-----------------------------*/
ShowCRC()
{
    char    *CRCstr = "    ";
    char    *ErrorMsg = "ProDOS Error!  Code ";
    Word    Error;

    WaitCursor();
```

```
DialogPort = GetNewModalDialog(&CRCBox);    /* Create modal dialog */
SetPort(DialogPort);

MoveTo(10,20);
DrawCString("Getting CRC on ");             /* Print a prompt */
DrawString(Reply.filename);
DrawCString("...");
MoveTo(BoxWidth/2 - 26, 36);

CRC = 0;                                    /* Init CRC at zero */
Error = GetCRC(Reply.fullPathname);         /* Get CRC on the file */
if (Error) {                                /* If an error occurred... */
    MoveTo(10,36);                          /* ...print a message */
    DrawCString(ErrorMsg);
    SysBeep(1);
    CRC = Error;
    }

    IntToHex(CRC, CRCstr + 1, 4);           /* Make CRC printable */
    DrawCString(CRCstr);                    /* Then print it */
    InitCursor();
    ModalDialog(NULL);                      /* Wait for OK button */
    CloseDialog(DialogPort);                /* Close the dialog */
}


/*-----------------------------*
 *      Shutdown Toolset       *
 *-----------------------------*/

ShutDownTools(
```





```
    SFShutDown();
    LEShutDown();
    CtlShutDown();
    WindShutDown();
    EMShutDown();
    QDAuxShutDown();
    QDShutDown();
    MTShutDown();
    DisposeAll(MemID);
    MMShutDown(UserID);
    TLShutDown();
}


/*-----------------------------*
 *          Main               *
 *-----------------------------*/

/* Display a Standard File Operations "Get" Dialog and wait for a
 * file to be selected.  If Cancel is selected, the program quits.
 */

main()
{
    StartUpTools();
    do {
        SFGetFile(Center-130, 35, "\pCalculate CRC on:", 0L, 0L, &Reply);
        if (Reply.good)     /* If Open clicked ... */
            ShowCRC();      /* ... do the CRC      */
    } while (Reply.good);
```

```
    ShutDownTools(1);
    QUIT (&QParms);
}
```

## CRC.ASM

To complete the machine language version of this program, simply steal parts of the MODEL.ASM and other examples from this book which correspond to most of the routines in CRC.C. Program 14-2 and 14-3 are two new subroutines, GetCRC and CalcCRC in machine language.

**Program 14-2. Calculate CRC on a Buffer**

```
*------------------------------*
*  Calculate CRC on a Buffer   *
*------------------------------*

CalcCRC ldx     #0              ;init index into buffer
NxtByte shorta                  ;go to 8-bit accumulator
        lda     Buffer,x        ;grab a character
        inx                     ;bump index
        eor     CRC+1           ;fix high-order byte of CRC
        sta     CRC+1
        longa                   ;back to 16-bit accumulator

        ldy     #8              ;init shift counter
Shift   asl     CRC             ;shift CRC left once always
        bcc     Next            ;if bit 15 was clear, skip the XOR

        lda     CRC             ;XOR CRC with $1021
        eor     #$1021
        sta     CRC
Next    dex
        bne     Shift           ;do this 8 times
```





```
        dec     xferNum         ;More bytes to do?
        bne     NxtByte         ;yes if count not zero

        rts

CRC     ds      2               ;here's the CRC word
```

**Program 14-3. Get CRC on the File**

```
*------------------------------*
*   Get CRC on the File        *
*------------------------------*

GetCRC  _OPEN   OParms          ;Open the file
        bcs     StopErr         ;stop if error occurred

        moveword Oref,Rref      ;Copy reference number
RdLoop  _READ   RParms          ;Read data from file into Buffer
        bcs     ChkErr          ;error occurred -- check for EOF

        jsr     CalcCRC         ;no error, so update CRC
        bra     RdLoop          ;and go back to read more until EOF

ChkErr  cmp     #$4C            ;end of file error
        bne     StopErr         ;no, so return it

        lda     #0              ;flag no error -- EOF isn't fatal
StopErr sta     Error           ;save error return code
        CLOSE   OParms          ;close the file
        rts                     ;and return

Error   ds      2               ;error code location
```

```
Buffer  ds      BufferSize      ;data buffer

OParms  ANOP                    ;OPEN Parameter List
Oref    ds      2               ;open file reference number
        dc      i4'pathname'    ;pointer to pathname
        ds      4               ;address of I/O buffer

RParms  ANOP                    ;READ Parameter List
Rref    ds      2               ;reference number for reading
        dc      i4'Buffer'      ;pointer to data buffer
        dc      i4'BufferSize'  ;size of buffer
transNum ds     4               ;transfer count
```

## CRC.PAS

The GetCRC and CalcCRC routines in *TML Pascal* are shown in Program 14-4.

### Program 14-4. Calculate CRC on a Buffer

```
{*------------------------------*
 *  Calculate CRC on a Buffer   *
 *------------------------------*}

{ Note that the global variable CRC has a range of $0000..$FFFF }

PROCEDURE CalcCRC (bufPtr: Ptr; count: Integer);
VAR     x:      Integer;
        Data:   $0000..$FFFF;
BEGIN
    REPEAT
        Data := bufPtr^;
        FOR x := 1 TO 8 DO
            Data := BitSL (Data);
```





```
        CRC := BitXOR (CRC, Data);
        bufPtr := Pointer (LongInt (bufPtr) + 1);
        FOR x := 1 TO 8 DO
            IF CRC > $7FFF THEN
                CRC := BitXOR (BitSL (CRC), $1021)
            ELSE
                CRC := BitSL (CRC);
        Dec (count);
    UNTIL (count = 0);
END;


{*------------------------------*
 *      Get CRC on the File     *
 *------------------------------*}


FUNCTION GetCRC (pathname: StringPtr) : Integer;
VAR Error:  Integer;
    EOF:    Boolean;
    Buffer: Packed Array [0..BufferSize] of Byte;
    OParms: P16ParamBlk;
    RParms: P16ParamBlk;
BEGIN
    EOF := FALSE;
    OParms.Pathname2 := pathname;
    P16Open (OParms);
    Error := IOResult;
    IF Error = 0 THEN BEGIN
        RParms.refNum := OParms.refNum;
        RParms.dataBuffer := @Buffer[0];
        RParms.requestCount := BufferSize;
```

```
        REPEAT
            PloRead (RParms);
            Error := IOResult;
            IF Error <> 0 THEN
                BEGIN
                    EOF := TRUE;
                    IF Error = $4C THEN Error := 0;
                END
            ELSE
                CalcCRC(theBuffer^[0], RParms.transferCount);
        UNTIL EOF;
    END;
    PloClose (OParms);
    GetCRC := Error;
END;
```

## Disk Errors

Table 14-7 is a complete list of error codes that can be returned by the ProDOS 16 operating system. (See the "Checking for Errors" section in this chapter for details on how to detect and handle errors).

**Table 14-7. ProDOS 16 Error Codes**

| Number | Meaning |
|---|---|
| $00 | No error |
| $01 | Invalid call number |
| $07 | ProDOS is busy |
| $10 | Device not found |
| $11 | Invalid device request |
| $25 | Interrupt vector table full |
| $27 | I/O error |
| $28 | No device connected |
| $2B | Disk is write-protected |
| $2E | Disk switched, files open |
| $2F | Device not online |
| $30–$3F | Device-specific errors |





| Number | Meaning |
|---|---|
| $40 | Invalid Pathname |
| $42 | File control block table full |
| $43 | Invalid reference number |
| $44 | Path not found |
| $45 | Volume not found |
| $46 | File not found |
| $47 | Duplicate pathname |
| $48 | Volume full |
| $49 | Volume directory full |
| $4A | Version error |
| $4B | Unsupported storage type |
| $4C | EDF encountered, out of data |
| $4D | Position out of range |
| $4E | Access not permitted |
| $50 | File is open |
| $51 | Directory structure damaged |
| $52 | Unsupported volume type |
| $53 | Invalid parameter |
| $54 | Out of memory |
| $55 | Volume control block full |
| $57 | Duplicate volume |
| $58 | Not a block device |
| $59 | Invalid file level |
| $5A | Block number out of range |
| $5B | Illegal pathname change |
| $5C | Not an executable file |
| $5D | File system not available |
| $5E | Cannot deallocate /RAM |
| $5F | Return stack overflow |
| $60 | Data unavailable |

## Chapter Summary

The following functions are part of the Standard File Operations tool set, which is presented in this chapter:

**Function:** $0117
Name: SFBootInit
Initialize the Standard File tool set environment
Push: Nothing
Pull: Nothing
Errors: None
Comments: Applications do not make this call.

**Function:** $0217
Name: SFStartup
Starts up the Standard File Operations tool set
Push: User ID (W); Direct Page (W)
Pull: Nothing
Errors: None
Comments: Call this before using Standard File functions.

**Function:** $0317
Name: SFShutdown
Shuts down the Standard File tool set and frees some memory
Push: Nothing
Pull: Nothing
Errors: None
Comments: Call this when your application is done using Standard File Operations.

**Function:** $0417
Name: SFVersion
Get the current version of the Standard File tool set
Push: Result Space (W)
Pull: Version number (W)
Errors: None

**Function:** $0517
Name: SFReset
Reset the Standard File Operations tool set
Push: Nothing
Pull: Nothing
Errors: None

**Function:** $0617
Name: SFStatus
Determine if the Standard File Operations tool set is active
Push: Result Space (W)
Pull: Active flag (W)
Errors: None
Comments: The flag is 0 if false and nonzero if true.

**Function:** $0917
Name: SFGetFile
Lets the user choose a specific file from a dialog box
Push: X position of dialog box (W); Y position of dialog box (W); Pointer to dialog box title string (L); Pointer to filtering subroutine (L); Pointer to list of valid file types (L); Pointer to returned pathname record structure (L)





Pull: Nothing
Errors: None
Comments: Title string starts with a count byte. Calling of the filtering routine can be inhibited by using $00000000 as its address. The filtering routine should return via RTL. File type list starts with a count byte. Record structure returned is the following: Open Flag (W); File type (W); Auxiliary file type (W); filename (16 bytes); full pathname to file (129 bytes).

**Function:** $0A17
Name: SFPutFile
Lets the user choose a filename for saving information to disk
Push: X position of dialog box (W); Y position of dialog box (W); Pointer to dialog box title string (L); Pointer to string containing original filename (L); Maximum length of name (W); Pointer to returned pathname record structure (L)
Pull: Nothing
Errors: None
Comments: Returned record structure is the same as SFGetFile.

**Function:** $0B17
Name: SFPGetFile
Allows user to choose a filename from a custom dialog box
Push: X position of dialog box (W); Y position of dialog box (W); Pointer to title string (L); Pointer to filtering routine (L); Pointer to file type list (L); Pninter to dialog template structure (L); Pointer to modal dialog event handler (L); Pointer to returned pathname record structure (L)
Pull: Nothing
Errors: None
Comments: Same as SFGetFile except for the template pointer and modal dialog activity handler (see Dialog Manager section for details).

**Function:** $0C17
Name: SFPPutFile
Gives the user a custom dialog box to choose a filename for saving information to disk.
Pointer to dialog box title string (L); Pointer to string containing original filename (L); Maximum length of name (W); Pointer to dialog template structure (L); Pointer to modal dialog event handler (L); Pointer to returned pathname record structure (L)

Pull: Nothing
Errors: None
Comments: See SFPGetFile.

**Function:** $0D17
Name: SFAllCaps
Sets the case mode for filenames in dialog boxes
Push: Case flag (W)
Pull: Nothing
Errors: None
Comments: If case flag is true (nonzero), all filenames in SFO dialog boxes will be shown without conversion to lowercase.



# Appendices

## Appendix A

# Apple's Human Interface Guidelines

The uniting idea behind the Macintosh and Apple IIGS desktop, windows, menu bars, icons, and dialog boxes is to give all software applications a universal look and feel. Apple wants its computers to be easy to learn and to use. To accomplish this, all software should follow the same conventions and use the same or similar methods of accomplishing many tasks.

> Witness the rabble of MS-DOS software, with its many programs and varying uses of graphics, the keyboard, and other conflicting methods of operating a program. The Human Interface Guidelines provide sanity and order in an operating system that might otherwise be just as confusing as the rest.

Contrary to what you've read, following the guidelines is not called *user-friendly* programming. Instead, Apple refers to it as *user-centered* programming. Most programs are written by programmers who wish to amaze other programmers. As a programmer yourself, you've probably been frustrated with the way things are supposed to be done using the desktop interface. After all, wouldn't it be much easier and faster to type an MS-DOS–like command such as COPY A:*.* C:\ROOT\DEV /B?

Perhaps you have noticed that the user interface of many non-Apple programs is poorly thought out. Among the dozens of word processors available for MS-DOS computers, there are radically different procedures to perform the same tasks. Some word processors have their own conventions and, for the convenience of users, allow alternate keypresses to mimic other word processors. Some even have vastly different sequences of commands within a single program to achieve similar results. This is the sort of disarray that naturally occurs when there is no enforced standard.

Apple has worked on its Human Interface Guidelines for years. The idea behind the guidelines is to make all programs running on





Apple computers behave the same, or enough alike that you only need to learn one technique for accomplishing similar tasks in several programs.

This appendix presents certain ideas and philosophies of the Human Interface Guidelines. It was decided that these ideas should all be placed together here, rather than scattered throughout the book. After all, you are a programmer. And usually the last thing you'll consider is how the first-time user will feel about using your program. Now that you know how to program the Apple IIGS Toolbox, it's time you learned how to present it to the user.

Programs are not judged on speed alone. Many programmers pride themselves in writing fast, compact code. Getting the job done, and done quickly, is important. But magazine reviewers and software salespeople will not recommend programs that don't follow these guidelines.

> Reading the Human Interface Guidelines is like reading a dog-eared, highlighted college text. The list is full of interesting ideas, thoughts, and reasons explaining why Apple did what it did in designing the Macintosh/Human interface.
>
> This book (and its predecessor, *COMPUTE!'s Mastering the Apple IIGS Toolbox*) constantly reminds you to "follow the conventions" and "do it this way." If you don't follow the guidelines, you may find your work incompatible with future releases of the computer or operating system.
>
> You'll notice that few programmers obey all of the rules and suggestions mentioned in the guidelines. Just as with other aspects of life, some people don't pay attention. Apple Computer itself is one of the worst offenders and doesn't always pay attention to those warnings, either. Just ask anyone who owns a Mac II. Because Apple didn't follow its own rules, a good deal of its own software won't work on the Mac II.
>
> If you buy and read a copy of the Human Interface Guidelines, you'll notice that there are many recommendations that Apple never follows. The best advice is to do what they say and not what they do. Follow the guidelines and you will avoid trouble in the future.

## What Are the Human Interface Guidelines?

Addison-Wesley has published a book written by Apple entitled *Human Interface Guidelines: The Apple Desktop Interface*. You can buy this book at your favorite bookstore (ISBN 0-201-17753-6). It's the latest rendition of an on-going project at Apple. While researching *Advanced Programming Techniques for Mastering the Apple IIGS Toolbox*, we located and mulled over one of the photocopied originals of the Human Interface Guidelines. Not much has changed since then. Only the list of contributing authors has grown longer. Still, most of the work can be attributed to Bruce Tognazzini (also lovingly called "Saint" Tognazzini). And before that, much of the philosophy on the interface came from work originally done at Xerox's Palo Alto Research Center (Xerox PARC).

Most of the beginning of the book is devoted to philosophizing and self-admiration of the Macintosh, mouse, and the desktop interface. Since you know how to point, click, and drag, that information is left out of this appendix.

Instead, you'll find the high points of the Human Interface Guidelines, all you really need to keep in step with what Apple likes to see in Apple programs. If you follow these guidelines, your program will be more compatible with other Apple IIGS and Macintosh programs. And Apple will like you. What more could you want?

## The Desktop Environment

The desktop environment is the latest, supposedly best way for a computer to communicate with a human. It's called *visual communication*. Rather than typing names and commands, you do things visually with the mouse and with graphic icons which appear on the screen.

You might think that this setup would mean anyone could use an Apple computer immediately. You would be wrong. People still have hang-ups about computers. No matter how easy you make them, some people would have you throw pitchforks at them before they would use a computer.

The following are highlights of the guidelines:

- Every action on the desktop should be as simple and consistent as possible. The Human Interface Guidelines give the greatest weight to visual communication, simplicity, and clarity.





- Don't be rude to the user. Always provide a way out. When you are given the choice between doing something potentially dangerous and backing out, always make the default choice the way out. In other words, it should never be easy to do something stupid.
- Keep your desktop consistent. Changing screen modes is about the most unforgivable offense. True, the 320 mode is more colorful, and the 640 mode can display more text. Yet a word processor that uses one mode for one thing and the second mode for another would be dreadful. Users admire stability.
- Cut down on the dazzle. You can do amazing things with the Toolbox and QuickDraw, but try not to overwhelm the user with spectacular graphics and stereophonic sound. Look up the word *aesthetic* in the dictionary if you have trouble with what programmers call *creeping elegance*.

## Programming for the Toolbox

In case you haven't noticed, all programs written for the Apple IIGS Toolbox follow a convention. They consist of a main event loop nested between setup and shutdown routines. (See Chapter 3 of *COMPUTE!'s Mastering the Apple IIGS Toolbox* for additional information.) This technique makes for better organization of your programs, making your programs easier to modify, as well (and incidentally, the code is easier to adapt for your other programs).

The following are a few concepts to keep in mind while designing and writing your programs:

- Implement what Apple calls *User Control* in your programs. Make the user choose what goes on. Don't make it appear that there is no way to control what the computer is doing.
- Provide the user with a complete list of options at any decision point. This is what separates desktop programs from IBM-type programs. In the IBM (command line interface) version of a program, it's up to you to remember what commands to type. With an Apple program, the user should see all the options available and then visually select one. Avoid hidden or secret options.
- When using an icon as a switch, make the icon closely resemble the action it invokes. For example, icons of an ImageWriter and LaserWriter can be used to choose a printer instead of an input box with the prompt *Enter printer:*.

- When you provide text (to explain a dialog box or amplify a choice, for instance), write a solid, meaningful description. Too many programmers opt to be overly cryptic with their text descriptions. But don't be overly simple with your text, either. Notice how *Send the contents of your document to the printer* is too simple, and *Dump File to Printer Device* is too complex, but *Print document: Chapter One?* is just right.

### Mouse Traps

The guidelines go into great detail about use of the mouse, to the extent of discussing the algorithms used to select text with the mouse. Since this is an internal function of the Toolbox there's no need to repeat it here. Instead, the following are the mouse highlights of the guidelines:

- Using the mouse with your programs should be consistent with other desktop programs. Remember the standard mouse operations (pointing, clicking, dragging, double-clicking, and so on). Don't make up new mouse modes that could confuse the user.
- Though all Apple computers now have cursor-control keys on their keyboards, Apple demands that you never use the arrow keys as a replacement for the mouse. Never. You shouldn't even use the arrow keys to choose menu selections. (It should be pointed out, however, that Apple uses the cursor keys to imitate the mouse on the Macintosh.)

Though you can change the cursor's shape to just about anything (a pointing finger, for example), the following shapes are suggested for certain activities:

**Figure A-1. Mouse Pointer (Cursor) Shapes**






- The pointer is the most common default cursor.
- The I-beam is used for inserting and selecting text. The I-beam (or, if active, the pointer), disappears when the user starts typing.
- The crosshairs pointer is used to select graphic shapes for manipulation.
- The plus sign is used in some spreadsheet programs to select cells in the worksheet. It can also be used to select fields in an array. (The original Macintosh spreadsheet program, *Multiplan*, first employed the plus sign.)
- The tiny wristwatch stands for a pause as the machine does some work behind the scenes.

### Pull-Down Menus

Menus are among the prime ingredients of the desktop. You should already know about menu titles and menu items and where they fit into the big picture. Keep in mind that the organization of menus and menu items (and command areas) is in your control.

### Standard Menus

There are three menus most programs should have. For the sake of consistency, certain menu items should appear only in these menus. The standard menus are

- The Apple menu
- The File menu
- The Edit menu

Text-based programs can also have Font, Style, and Size menus. However, Apple is less fussy about them.

- The Apple menu is always the first menu on the far left side of the menu bar. The first item at the top of this menu is an About. . . menu item used to display a dialog box telling about your program.
- Under the About. . . item come the various desk accessories installed in your SYSTEM/DESK.ACCS subdirectory. Also, you can put a Help item in the Apple menu and any configuration item or desk accessories specific to the application, such as a spelling checker for a word processor.
- The File menu contains all the items that deal with saving, loading, and creating data files. Aside from its allowances for opening, closing, and saving files, this menu also contains print options

and the Quit option. Even if your program lacks any disk access, this is where the Quit option should go. The typical file menu appears as shown in Figure A-2.

**Figure A-2. Graphic of File Menu**


- The final required menu is the Edit menu. A lot of emphasis is put on the cut-and-paste aspect of the desktop. Therefore, the Edit menu is considered important to all applications. (Even if your program doesn't need the items in the Edit menu, you might want them included for use by some desk accessories.)

**Figure A-3. Graphic of Edit Menu**


One of the common items on the Edit menu is Select All. There is no key equivalent officially defined for the Select All item, although many applications seem to implement their own (usually Open Apple–A).

Other menus that might crop up from time to time, especially in text-oriented programs, are

- The Font menu
- The Size menu
- The Style menu





There are no hard-and-fast guidelines for these menus. If you have a crowded menu bar, you can combine two or three of them into one menu, or include all the options in a dialog box that looks like a Boeing 747 control panel.

### Menu Items

The guidelines have the following suggestions for menu items:

- Menu items can be verbs used to describe an immediate reaction, or they can be adjectives used to describe some attribute of a selection:
  - With verb menu items, you choose a menu item and that task is carried out. If the program requires more input before the action can take place, the menu item should be followed by ellipses (. . .). If the item toggles a state on or off, a check mark should appear to indicate when the item is on, or you can choose to change the menu item's text—for example, *Inhale* could be changed to *Exhale*.
  - When menu items are adjectives (in font menus, for example), they should be descriptive words and adequately characterize what they change. Consider the opaqueness of a menu entry such as *Font 2* when it is compared to something more descriptive, such as *Courier*.

A third type of menu, introduced with the Apple IIGS, is the color menu. This menu has no words, only the hues of colors available for changing selected items.

Commonly used menu items should be at the top of a menu, with less frequently used items at the bottom. Good examples are the Undo menu item commonly found at the top of the Edit menu, and the Quit menu item found at the bottom of the File menu.

### Key Equivalents

You can assign key equivalents to just about any menu item. Be sure they make sense. Also consider that some actions are appropriate for the mouse and others are appropriate for the keyboard. A word processor is keyboard-intensive (although a mouse is great for editing). When users are typing, they'll find it more convenient to use a keyboard equivalent of a pull-down menu item than to grab the mouse, pull down the menu, and make the selection. On the other side of the coin, a paint program is a mouse-intensive piece of software. Having a keyboard-only command could be awkward.

Some of the older Macintosh communications programs lacked key equivalents entirely. This was because they used the Command key (later the Open Apple key) instead of the Control key to generate control codes. Apple IIGS communications programs have access to both the Open Apple–Command key and the Control key. So there is no reason to write a program devoid of Apple key equivalents.

The following keyboard equivalents must be used exclusively for the function described. Aside from these, you can assign whatever key equivalents your program might require:

| Keyboard Equivalent | Comment | Menu |
|---|---|---|
| Open Apple–? | Help | Apple |
| Open Apple–C | Copy | Edit |
| Open Apple–N | New | File |
| Open Apple–O | Open | File |
| Open Apple–Q | Quit | File |
| Open Apple–S | Save | File |
| Open Apple–V | Paste | Edit |
| Open Apple–X | Cut | Edit |
| Open Apple–Z | Undo | Edit |

Note that Open Apple–/ is considered the same as Open Apple–? (which is actually Open Apple–Shift–/). (See Chapter 8 for information on defining these keys.)

Less stringently obeyed are the following text-style key equivalents:

| Keyboard Equivalent | Comment |
|---|---|
| Open Apple–B | Bold |
| Open Apple–I | Italic |
| Open Apple–P | Plain |
| Open Apple–U | Underline |

A special-case Open Apple key equivalent is Open Apple–. (Open Apple–period). This key equivalent can be used to halt an action such as printing a document or a file listing in the APW shell. Apple implemented Open Apple–. because some Macintosh keyboards lacked an Esc key (normally Esc would be used). A few older programs may stick with the Open Apple–. convention. However, if you decide to implement an Esc cancel key in your programs, you might want to add Open Apple–. just to be well-received.





## Dialog Boxes

Dialog boxes are actually special forms of windows. They are divided into modal and modeless types, as well as the special-case Alert boxes. The guidelines include the following information about dialog boxes:

- Dialog boxes should be placed in the center of the upper third of the screen. (The examples in this book were positioned in the center of the upper half because it was more aesthetically pleasing.)
- Alert boxes can be positioned so that their default button is in the same position as that occupied by the button that activated them. For example, this would allow the user to quickly cancel an operation without moving the mouse.
- A dialog box should always contain a message. It might describe what the dialog does or give some indication of what is happening. Don't crowd the text into the dialog. If you need more room, make the dialog box bigger.
- The most important and most commonly used items in a dialog box should be placed at the top, just as they are in the pull-down menu. Less frequently used items should be placed at the bottom. You can also place the more important items on the left side of the box, and the less important ones on the right.
- Remember to include in the dialog box a button that lets the user out.
- The OK button is associated with the Return key and the Cancel button is associated with the Esc key. Don't confuse the user by mixing these up.

There is such a creature as a dialog box without buttons. An example is a simple text box that displays a message and then disappears. One use for this sort of dialog is to inform the user how long an operation will take. For some reason, users don't mind waiting 15 minutes for an operation if the program is smart enough to tell them to do so.

## Alerts

An alert dialog box is an example of a specific dialog with a specific use. In some cases, you may find that a simple beep of the speaker will replace an alert. For example, if a user clicks outside of a field, it's much faster to make the speaker *bonk* than to bring up a complete alert box.

Take advantage of the various alert stages. During your beta testing, you may discover that some alert boxes appear more often than others, indicating perhaps that a specific type of error is more likely to occur. If so, you may want to rethink your program's strategy. Ask questions of your beta testers to see if this happens.

The guidelines make the following suggestions:

- Keep alerts clean. Don't use radio buttons, long-scrolling text messages, check boxes, or other clutter. The typical alert box has an alert icon, a short message (or warning), and two buttons.
- The two buttons in an alert box typically allow the chosen action to continue or to be stopped. For example, an alert might display the message *Erase your hard disk?* The two buttons could be *Yes* and *No*, or even better, *Erase* and *Stop*. Typically, however, you should phrase your prompts so that it would be natural to supply buttons marked *OK* and *Cancel*.
- The default button in an alert box is always Cancel. The purpose of the alert is to warn of some impending danger. The default choice should always be to back away from the danger—in other words, make users really think about what they're doing.
- The alert message could be a system error, or something that your program can't handle. When this is the case, you may want to rethink your error-trapping routines and perhaps take the error-correcting decision out of the user's hands.

## Notes on Sound and Color

The Apple IIGS comes with excellent sound and graphics. With the addition of the Mac II, sound and color have also been made available to the Macintosh line of computers.

The following are the guidelines on the use of sound and color in your programs. Generally speaking, the suggestions themselves are rather obvious, if you think about them. Listed below are only the high points.

**Sound.** The general thrust of the guidelines approach to sound is that sound should be used as an attention-getter. Use sound to say *Hey you!* should an application require immediate attention, or use it to alert the user that something is happening in the background. Other highlights:

- Try not to startle the user with sound.





- Different sounds can be used to herald entering and exiting certain modes. Of course, you may find these modal sounds annoying. It would be nice to include an option in your program for shutting off the noises (or for a volume control, at least).

**Color.** Color can be fun, and a great benefit to your programs. However, there are a few guidelines about the use of color. Most of these you can figure out on your own. For instance, an all-red foreground and background can make computing difficult. Still, some of the other guidelines are interesting and, when you pause to think about them, make sense.

- Different colors can be used in a number of ways. For example, you can color some text or a dialog box icon red to indicate something drastic. The color yellow can be used to show caution. Green is used to indicate *go* or *proceed*.
- Blue, especially light blue, is hard to see, and the guidelines recommend avoiding its use. However, an example of a good use of light blue would be providing rules or grids for a paint program; the blue is just faint enough to use as a reference.
- Use color to show how certain objects are grouped together, or to define separate areas.
- Keep the background light. A dark red background will make any foreground text difficult to see. Some programmers get carried away with color. Remember that users just want to use your program. Psychedelic colors went out with the sixties, along with love beads and sandalwood incense.

Above all, consider the application. Colored text looks good on the screen (and has probably sold more than one Apple IIGS). However, few people can print colored text. If the application is one that could use some color—such as a drawing, painting, or educational program—use it. But for text-intensive programs, think twice before splashing the screen with color, or at least provide the user with the option of choosing the colors to be used on the text display.

It should be noted that the terms *text* and *text display* have been tossed about freely in this appendix. True, the Apple IIGS does have a text display mode that can use different-colored backgrounds and letters. But the references to text *are meant* to include any textual material displayed on the graphics screen as well.

### Summary

It goes without saying that a copy of the Apple Human Interface Guidelines will provide more detailed information than this appendix. However, the desktop environment is constantly changing. As Apple develops the IIGS and its other computers, and as programmers provide more interesting and intuitive applications, the guidelines will no doubt change. Just remember these two things:

- Users love to play with things.
- Above all, have fun with your programming.



## Appendix B

# Tool Sets in the Apple IIGS Toolbox

**Table B-1. Tool Sets**

| Number | Tool Set Name | Version |
|---|---|---|
| $01 | Tool Locator | $0201 |
| $02 | Memory Manager | $0200 |
| $03 | Miscellaneous | $0200 |
| $04 | QuickDraw II | $0202 |
| $05 | Desk Manager | $0202 |
| $06 | Event Manager | $0201 |
| $07 | Scheduler | $0200 |
| $08 | Sound Manager | $0201 |
| $09 | Apple DeskTop Bus | $0201 |
| $0A | SANE | $0202 |
| $0B | Integer Math | $0200 |
| $0C | Text Tool Set | $0200 |
| $0D | RAM Disk | $0200 |
| $0E | Window Manager | $0201 |
| $0F | Menu Manager | $0200 |
| $10 | Control Manager | $0202 |
| $11 | System Loader | —?— |
| $12 | QuickDraw II Auxiliary | $0202 |
| $13 | Print Manager | $0102 |
| $14 | LineEdit | $0200 |
| $15 | Dialog Manager | $0200 |
| $16 | Scrap Manager | $0102 |
| $17 | Standard File | $0200 |
| $18 | Disk Utilities | —?— |
| $19 | Note Synthesizer | $0100 |
| $1A | Note Sequencer | —?— |
| $1B | Font Manager | $0201 |
| $1C | List Manager | $0201 |

The high-order byte of the version number indicates the major release number and the low-order byte is the minor release. If bit 7 of the major release is set (bit 15 of the word), the release is a beta

version. For example, $0201 (binary 0000 0010 0000 0001) indicates version 2.1, and $8101 (binary 1000 0001 0000 0001) indicates beta (prerelease) version 1.1.

The version numbers above apply to the ROM 01 release of the Apple IIGS as well as to the tool sets on System Disk version 3.1. Version numbers shown as —?— indicate tool sets which are not yet available.

## TV.C Program

Since version numbers change as fast as the wind in Cupertino, Program B-1 (a C program) will generate a table just like the one printed above with the latest tool set version information for your system.

Program B-1. TV.C

```
/*-------------------------------------------*
 *                   TV.C                    *
 *    Displays all known toolset versions    *
 *-------------------------------------------*/

#include <types.h>
#include <prodos.h>
#include <intmath.h>
#include <locator.h>
#include <memory.h>
#include <misctool.h>
#include <texttool.h>
/*-----------------------------*
 *          Main               *
 *-----------------------------*/

Word UserID;            /* Our User ID */
Word Toolist[] = {
        12,             /* Tool count */
```





```
        14, 0,          /* Window Manager */
        15, 0,          /* Menu Manager */
        16, 0,          /* Control Manager */
        18, 0,          /* QD II Aux */
        19, 0,          /* Print Manager */
        20, 0,          /* Line Edit */
        21, 0,          /* Dialog Manager */
        22, 0,          /* Scrap Manager */
        23, 0,          /* Standard File */
        25, 0,          /* Note Synth */
        27, 0,          /* Font Manager */
        28, 0           /* List Manager */
};

QuitRec QParms = { NULL, 0 };   /* ProDOS 16 Quit parameter list */

main()
{
   TLStartUp   ();                      ErrChk();
   UserID      = MMStartUp (1);         ErrChk();
   MTStartUp   ();                      ErrChk();
   WriteString ("\pLoading tools...");
   LoadTools   (Toolist);               ErrChk();
   WriteLine   ("\p");
   WriteLine   ("\p");

   ShowVers    ();                      /* Show Versions */

   MTShutDown  ();
   MMShutDown  (UserID);
   TLShutDown  ();
```

```
    QUIT        (&QParms);              /* Quit to ProDOS */
}

/*-----------------------------*
 *    Handle Toolbox Errors    *
 *-----------------------------*/

ErrChk() {  if (_toolErr) SysFailMgr(_toolErr, NULL);  }

/*-----------------------------*
 *    Show Toolset Versions    *
 *-----------------------------*/
 struct set {
     char    *name;
     word    id;
 } Toolset[] = {
     "\pTool Locator",          1,
     "\pMemory Manager",        2,
     "\pMiscellaneous Tools",   3,
     "\pQuickDraw II",          4,
     "\pDesk Manager",          5,
     "\pEvent Manager",         6,
     "\pScheduler",             7,
     "\pSound Manager",         8,
     "\pApple Desktop Bus",     9,
     "\pSANE",                  10,
     "\pInteger Math",          11,
     "\pText Toolset",          12,
     "\pRAM Disk",              13,
     "\pWindow Manager",        14,
     "\pMenu Manager",          15,
```





```
    "\pControl Manager",       16,
    "\pSystem Loader",         17,
    "\pQuickDraw II Aux.",     18,
    "\pPrint Manager",         19,
    "\pLineEdit",              20,
    "\pDialog Manager",        21,
    "\pScrap Manager",         22,
    "\pStandard File",         23,
    "\pDisk Utilities",        24,
    "\pNote Synthesizer",      25,
    "\pNote Sequencer",        26,
    "\pFont Manager",          27,
    "\pList Manager",          28
};

#define ENTRIES (sizeof (Toolset) / sizeof (struct set))

char    *HexStr =   "\p$xxxx";
char    *Title  =   "\pNo.  Toolset Name         Version     ";
/*                   ;****;****;****;****:****:****;****:**** */

ShowVers()
{
    word    i;

    WriteString (Title);    WriteLine (Title);
    RepeatChar  ('=', 71);  WriteLine ("\p");

    for (i = 0; i < ENTRIES/2; ++i) {
        DoLine (i);
```

```
        RepeatChar (32, 6);
        DoLine (i + ENTRIES/2);
        WriteLine ("\p");
    }
}

DoLine(item)
word item;
{
    word    id, ver;

    id = Toolset[item].id;
    Int2Hex (id, HexStr+2, 2);
    HexStr[5] = HexStr[4] = 32;
    WriteString (HexStr);
    WriteString (Toolset[item].name);
    RepeatChar (32, 22 - (Toolset[item].name[0] & 0xff));
    asm {
        lda id
        ora #1024
        tax
        pha
        jsl dispatcher
        sta _toolErr
        pla
        sta ver
    }
    if (_toolErr)
        WriteString ("\p--?--");
    else {
```





```
        Int2Hex     (ver, HexStr+2, 4);
        WriteString (HexStr);
    }
}

RepeatChar (theChar, count)
char    theChar;
int     count;
{
    while (count--) WriteChar (theChar);
}
```

This program makes a handy utility to keep around on your *APW* system disk. To direct its output to a file or printer, use one of these *APW* shell commands:

```
tv >filename
tv >.printer
```

```
        Int2Hex     (ver, HexStr+2, 4);
        WriteString (HexStr);
    }
}

RepeatChar (theChar, count)
char    theChar;
int     count;
{
    while (count--) WriteChar (theChar);
}
```

This program makes a handy utility to keep around on your *APW* system disk. To direct its output to a file or printer, use one of these *APW* shell commands:

```
tv >filename
tv >.printer
```





# Error Handling

There are several ways to deal with errors returned from the Toolbox. A blanket method has been shown in this book, one that's not really the best way to deal with potential errors. In fact, the method used by most Toolbox program examples in this book would be considered awful error trapping for a professional application.

Most of your programs should be smart enough to catch simple, common errors. Out-of-memory errors, disk I/O errors, and some Toolbox errors can easily be sidestepped. Your programs should make exceptions for the errors, recognize them, and deal with them in such a manner as to be transparent to the user. In other words, don't cop out on error handling.

## ErrChk

Program C-1 is the error-checking code used in this book as the generic error handler, ErrChk. The problem with ErrChk is that it assumes every error returned from the Toolbox is a fatal, typically death-inducing error.

**Program C-1. ErrChk in Machine Language**

```
*------------------------------*
#   Handle Toolbox Errors      #
*------------------------------*

ErrChk  bcs Die                 ;Carry set if error
        rts                     ;Else, return

Die     pha                     ;Toolbox returns error in A
        pushlong #0             ;Use standard system death message
        _SysFailMgr             ;Get ready to slide apples back and forth
```

Program C-2 is the equivalent in C.

**Program C-2. ErrChk in C**

```
/*-----------------------------*
 *    Handle Toolbox Errors    *
 *-----------------------------*/


ErrChk()                            /* Check for error, die if so */
{
  if (_toolErr) SysFailMgr(_toolErr, nil);
}
```

Program C-3 is the equivalent in Pascal.

**Program C-3. ErrChk in Pascal**

```
{ *-----------------------------*
  *    Handle Toolbox Errors    *
  *-----------------------------* }


PROCEDURE ErrChk;                   { Check for error, die if so }
BEGIN
  IF IsToolError THEN
      SysFailMgr(ToolErrorNum, StringPtr(0));
END;
```

This error handler is called after every potential error-causing Toolbox function. All it checks is whether an error occurred. If so, the program bombs using the SysFailMgr call and tells you there's a fatal error. This is a very nondescript and somewhat crude method of error handling, albeit good for quick demonstrations and beta testing. But it doesn't take into consideration errors from which recovery is possible.





## A Better Generic Error Handler

Documenting a procedure for each nonfatal Toolbox error would be complicated and would increase the size of this appendix to a full-blown chapter. Instead, the following example is provided to pique your curiosity.

This error-trapping routine (Program C-4) is designed to handle generic errors, and it can replace the ErrChk routine used throughout this book. Of course, it's a good idea to take care of nonfatal errors individually. This routine should be called only as a last-ditch effort. The code is listed in machine language. C and Pascal programmers can be inventive and code their own versions.

**Program C-4. Fatal Error Handler**

```
*-----------------------------*
*    Fatal Error Handler      *
*-----------------------------*

;only absolutely fatal errors are sent here

Die     pha                     ;Toolbox returns error in A, save it
        and #$FF00              ;get toolset number
        xba                     ;exchange MSB half of A-reg to LSB
        clc                     ;clear carry
        tay                     ;put a into Y
        lda #Table-28           ;offset from start of table
Die0    adc #28                 ;length of each entry
        dey                     ;dec count
        bne Die0                ;loop until the toolset is indexed

        phb                     ;push data bank twice
        phb                     ;(because phb pushes only a byte)
        pha                     ;push string's address
        _SysFailMgr             ;Get ready to slide apples back and forth
```

```
Table   str '       Tool Locator error $'
        str '     Memory Manager error $'
        str 'Miscellaneous Tools error $'
        str '       QuickDraw II error $'
        str '       Desk Manager error $'
        str '      Event Manager error $'
        str '          Scheduler error $'
        str '      Sound Manager error $'
        str '  Apple Desktop Bus error $'
        str '              SANE error $'
        str '       Integer Math error $'
        str '       Text Toolset error $'
        str '           RAM Disk error $'
        str '    Window Manager error $'
        str '       Menu Manager error $'
        str '    Control Manager error $'
        str '      System Loader error $'
        str '  QuickDraw II Aux. error $'
        str '      Print Manager error $'
        str '           LineEdit error $'
        str '     Dialog Manager error $'
        str '      Scrap Manager error $'
        str '      Standard File error $'
        str '     Disk Utilities error $'
        str '   Note Synthesizer error $'
        str '     Note Sequencer error $'
        str '       Font Manager error $'
        str '       List Manager error $'
```





This routine eliminates the *Fatal System Error* message and replaces it with something more specific. Rather than providing a two-byte hex number, this example translates the first number, representing the tool set, into a string. The actual error number is displayed after the dollar sign. So instead of

```
Fatal System Error —> $0E02
```

you are given

```
Window Manager error $0E02
```

Granted, this routine doesn't do anything the standard ErrChk routine didn't do, but it's more specific as to the type of error occurring. Again, a specific routine to deal with certain types of errors would be better.

This routine is still relatively simple. It would be easy to make it more elegant. For example, rather than padding each error string so that each takes up a fixed number of characters, you could use a table of pointers into variable length strings. It takes more source code to implement, but results in far less object code.

## Appendix D

# Error Codes

There are three types of errors you can receive from your computer. Unfortunately, it's sometimes hard to determine the origin of an error, though this appendix should help. The three types of errors you can receive are

- Fatal System errors
- ProDOS errors
- Toolbox errors

Fatal System errors are errors your programs won't be able to catch or wouldn't want to catch. Because these errors seem to pop up quite often for adventurous programmers such as yourself, they're listed here.

ProDOS errors are different from Toolbox errors in that their origins are in ProDOS and are not the result of any Toolbox mistakes you might have made. In fact, anyone who has programmed disk I/O or worked at all with any operating system is familiar with a DOS error. You can't build a decent program without DOS error trapping.

ProDOS errors are not incurable. For example, if your program returned the error *Disk Write Protected*, you could prompt the user to remove the write-protect tab or use another disk.

Toolbox errors aren't always fatal. In fact, quite a few are survivable (see Appendix C). However, more often than not, your program's error-handling routine may report a few of the more interesting ones. If your error-handling routine is smart, it can work around the error. Otherwise, make sure your program displays the error code so your users can report it back to you.

> Just to throw you a curve, there are some Toolbox function calls that result in errors originating from ProDOS. Yes, it's true. For example, the Tool Locator's LoadTools call or the Font Manager's FMStartUp function can return with an error flagged. An error code between $0001 and $00FF is a ProDOS 16 error. Error codes greater than $00FF are Toolbox errors.





**Table D-1. Fatal System Errors**

| | |
|---|---|
| $01 | Unclaimed interrupt |
| $0A | Volume control block unusable |
| $0B | File control block unusable |
| $0C | Block 0 allocated illegally |
| $0D | Interrupt occurred while I/O shadowing off |
| $11 | Wrong OS version |

**Table D-2. Errors Returned from ProDOS**

| | |
|---|---|
| $00 | No error |
| $01 | Invalid call number |
| $07 | ProDOS is busy |
| $10 | Device not found |
| $11 | Invalid device request |
| $25 | Interrupt vector table full |
| $27 | I/O error |
| $28 | No device connected |
| $2B | Disk is write-protected |
| $2E | Disk switched, files open |
| $2F | Device not online |
| $30–$3F | Device-specific errors |
| $40 | Invalid pathname |
| $42 | File control block table full |
| $43 | Invalid reference number |
| $44 | Path not found |
| $45 | Volume not found |
| $46 | File not found |
| $47 | Duplicate pathname |
| $48 | Volume full |
| $49 | Volume directory full |
| $4A | Version error |
| $4B | Unsupported storage type |
| $4C | EOF encountered, out of data |
| $4D | Position out of range |
| $4E | Access: file not rename-enabled |
| $50 | File is open |
| $51 | Directory structure damaged |
| $52 | Unsupported volume type |
| $53 | Invalid parameter |
| $54 | Out of memory |
| $55 | Volume control block full |
| $57 | Duplicate volume |
| $58 | Not a block device |

| | |
|---|---|
| $59 | Invalid file level |
| $5A | Block number out of range |
| $5B | Illegal pathname change |
| $5C | Not an executable file |
| $5D | File system not available |
| $5E | Cannot deallocate /RAM |
| $5F | Return stack overflow |
| $60 | Data unavailable |

**Table D-3. Errors Returned from the Toolbox**

| | |
|---|---|
| $0000 | No error |
| $0001 | Internal error, not enough arguments on the stack |
| $0002 | Tool set wasn't activated (no StartUp call was made) |
| $0100 | Unable to mount system startup volume |
| $0110 | Bad tool set version number |
| $0201 | Unable to allocate block |
| $0202 | Illegal operation on an empty handle |
| $0203 | Empty handle expected for this operation |
| $0204 | Illegal operation on a locked or immovable block |
| $0205 | Attempt to purge an unpurgeable block |
| $0206 | Invalid handle given |
| $0207 | Invalid User ID given |
| $0208 | Operation illegal on block specified attributes |
| $0301 | Bad input parameter |
| $0302 | No device for input parameter |
| $0303 | Task is already in the heartbeat queue |
| $0304 | No signature in task header was detected |
| $0305 | Damaged queue was detected during insert or delete |
| $0306 | Task was not found during delete |
| $0307 | Firmware task was unsuccessful |
| $0308 | Detected damaged heartbeat queue |
| $0309 | Attempted dispatch to a device that is disconnected |
| $030B | ID tag not available |
| $0401 | QuickDraw already initialized |
| $0402 | Cannot reset |
| $0403 | QuickDraw is not initialized |
| $0410 | Screen is reserved |
| $0411 | Bad rectangle |
| $0420 | Chunkiness is not equal |
| $0430 | Region is already open |
| $0431 | Region is not open |
| $0432 | Region scan overflow |





| | |
|---|---|
| $0433 | Region is full |
| $0440 | Poly is already open |
| $0441 | Poly is not open |
| $0442 | Poly is too big |
| $0450 | Bad table number |
| $0451 | Bad color number |
| $0452 | Bad scan line |
| $0510 | Desk accessory is not available |
| $0511 | Window pointer does not belong to the NDA |
| $0601 | The Event Manager has already been started |
| $0602 | Reset error |
| $0603 | The Event Manager is not active |
| $0604 | Bad event code number (greater than 15) |
| $0605 | Bad button number value |
| $0606 | Queue size greater than 3639 |
| $0607 | No memory for event queue |
| $0681 | Fatal error: event queue is damaged |
| $0682 | Fatal error: event queue handle is damaged |
| $0810 | No DOC chip or RAM found |
| $0811 | DOC address range error |
| $0812 | No SAppInt call made |
| $0813 | Invalid generator number |
| $0814 | Synthesizer mode error |
| $0815 | Generator busy error |
| $0817 | Master IRQ not assigned |
| $0818 | Sound Tools already started |
| $08FF | Fatal error: unclaimed sound interrupt |
| $0910 | Command not completed |
| $0982 | Busy, command pending |
| $0983 | Device not present at address |
| $0984 | List is full |
| $0B01 | Bad input parameter |
| $0B02 | Illegal character in input string |
| $0B03 | Integer or long-integer overflow |
| $0B04 | String overflow |
| $0E01 | First word of parameter list is the wrong size |
| $0E02 | Unable to allocate window record |
| $0E03 | Bits 14–31 not clear in task mask |
| $1101 | Segment or entry not found |
| $1102 | Incompatible object module format (OMF) version |
| $1104 | File is not a load file |
| $1105 | System Loader is busy |

$1107 File version error
$1108 UserID error
$1109 Segment number is out of sequence
$110A Illegal load record found
$110B Load segment is foreign

$1401 The LEStartUp call has already been made
$1402 Reset error
$1404 The desk scrap is too big

$150A Bad item type
$150B New item failed
$150C Item not found
$150D Not a modal dialog

$1610 Unknown scrap type

$1B01 Font Manager has already been started
$1B02 Can't reset Font Manager
$1B03 Font Manager is not active
$1B04 Family not found
$1B05 Font not found
$1B06 Font is not in memory
$1B07 System font cannot be purgeable
$1B08 Illegal family number
$1B09 Illegal size
$1B0A Illegal name length
$1B0B FixFontMenu never called

$1C01 Unable to create list control or scroll bar control





# Event and TaskMaster Codes

Programs written for the Apple IIGS Toolbox center themselves on one event-oriented loop. Everything that happens in your programs is based upon a certain event—a mouse click, a drag, a selection. The Event Manager and its cousin the TaskMaster are at the heart of most DeskTop applications.

These events provide user input to your program. To determine which event has taken place (a mouse click, menu selection, or press of a key), your program makes a call to either the Event Manager's GetNextEvent function, or the Window Manager's TaskMaster function. Both of these procedures are covered within this book.

## The Event Manager

The primary function of the Event Manager is GetNextEvent:

**Function:** $0A06
Name: GetNextEvent
Returns the status of the event queue.
Push: Result Space (W); Event Mask (W); Event Record (L)
Pull: Logical Result (W)
Errors: None
Comments: If the Result is a logical true, an event is available. The event is then removed from the queue.

GetNextEvent deals with two items, the event mask and the event record. The event mask is used to scan only for specific types of events. The event record contains information about the event when GetNextEvent returns a logical true.

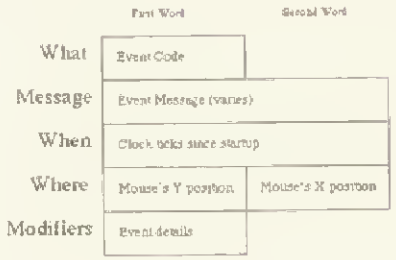
**The event mask.** The event mask is a word-sized value used to filter out certain types of events. By setting specific bits in the event mask, your program can direct GetNextEvent to return only the results of specific events. The following chart shows which bits in the event mask affect which events.

**Table E-1. Bit in the Event Mask**

| Bit | Events Scanned for, if Set |
|---|---|
| 0 | Not used |
| 1 | Mouse-down events |
| 2 | Mouse-up events |
| 3 | Keyboard (key-down) events |
| 4 | Not used |
| 5 | Auto-key events |
| 6 | Update events |
| 7 | Not used |
| 8 | Activate events |
| 9 | Switch events |
| 10 | Desk accessory events |
| 11 | Device drive events |
| 12 | User-defined events |
| 13 | User-defined events |
| 14 | User-defined events |
| 15 | User-defined events |

When GetNextEvent returns a true value, the event record will contain information detailing the event.

**Figure E-1. The Event Record**




The Event Record





The structure of the event record is as shown in Table E-2.

**Table E-2. Structure of Event Record**

| Field | Size | Description |
|---|---|---|
| What | Word | Code describing event |
| Message | Long | Value or pointer providing more detail about the event |
| When | Long | Number of clock ticks since the computer was started |
| Where | Long | Two word values; the Y and X position of the mouse at the time of the event |
| Modifiers | Word | Describes the state of certain keys, the mouse button, and other information |

**What.** The What field contains the event code. This describes which event took place. The events are numbered 0–15 (these are not bit values). The value found in the What field will be one of those shown in Table E-3.

**Table E-3. Events in What Field**

| Event Code | Description |
|---|---|
| 0 | Null Event: Nothing has happened. |
| 1 | The mouse button was just pressed. |
| 2 | The mouse button has been released. |
| 3 | A key on the keyboard is being pressed. |
| 4 | Not used. |
| 5 | Auto-key event: a key is being held down. |
| 6 | Update event: a window is being changed, redrawn, sized, or its contents updated. |
| 7 | Not used. |
| 8 | Activate event: generated when a window becomes either active or inactive. |
| 9 | Switch event: activated when one program switches control to another. |
| 10 | Control–Open Apple–Esc has been pressed (this event is handled by the Desk Manager). |
| 11 | A device driver has generated an event. |
| 12 | User-defined (can be defined by your application). |
| 13 | User-defined. |
| 14 | User-defined. |
| 15 | User-defined. |

**Message.** The Message field's value depends on the event code found in the What field.

**Table E-4. Message Returned**

| Event Code | Message Field Contents |
|---|---|
| 0 | Undefined. |
| 1 | Button number (low-order word only). |
| 2 | Button number (low-order word only). |
| 3 | ASCII character (lowest byte only). |
| 4 | Undefined. |
| 5 | ASCII character (lowest byte only). |
| 6 | Window pointer. |
| 7 | Undefined. |
| 8 | Window pointer. |
| 9 | Undefined. |
| 10 | Undefined. |
| 11 | Value is returned from the device driver. |
| 12 | Value is returned from the user-defined application. |
| 13 | Value is returned from the user-defined application. |
| 14 | Value is returned from the user-defined application. |
| 15 | Value is returned from the user-defined application. |

**When.** The When field contains the number of clock ticks since the computer was started. Each tick equals 1/60 second.

**Where.** The Where field gives the location of the mouse pointer at the time of the event, even if the event isn't mouse-oriented. The first word of the Where field contains the mouse's Y (vertical) position; the second word contains the mouse's X (horizontal) position.

**Modifiers.** The Modifiers field allows further description of the event pulled from the event queue.

**Table E-5. Modifiers**

| Bit | Description |
|---|---|
| 0 | If set, the window pointed to in Message field is being deactivated; otherwise, the window is activated. |
| 1 | If set, the active window is changing from the system window to an application's window, or vice versa. |
| 2 | Not used. |
| 3 | Not used. |
| 4 | Not used. |
| 5 | Not used. |
| 6 | If set, mouse button number 1 is down. |
| 7 | If set, mouse button number 0 is down. |
| 8 | If set, the Open Apple key is down. |
| 9 | If set, a Shift key is down. |





| Bit | Description |
|---|---|
| 10 | If set, the Caps Lock key is down. |
| 11 | If set, the "option" (Solid Apple) key is down. |
| 12 | If set, the Control key is down. |
| 13 | If set, a key on the keypad is down. |
| 14 | Not used. |
| 15 | Not used. |

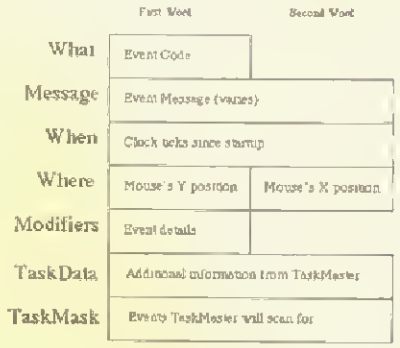
## TaskMaster

TaskMaster, though a function of the Window Manager, is similar to GetNextEvent. It adds extra functions for managing windows and pull-down menus and, secretly, calls GetNextEvent internally:

**Function:** $1D0E
Name: TaskMaster
Returns status of the event queue as well as checks for certain window/menu events.
Push: Result Space (W); Event Mask (W); Event Record (L)
Pull: Extended Event Code (W)
Errors: $0E03

TaskMaster uses the same event mask as described above. It adds, however, two fields to the event record, TaskData and TaskMask:

**Figure E-2. Event Record with TaskMaster Fields Added**




The Event Record plus TaskMaster Fields

**Extended event codes.** Unlike GetNextEvent, which returns a true or false value, TaskMaster returns either an event code or 0. When an event occurs, TaskMaster returns a value representing the event code. This code incorporates all the values found in the What field of the event record after a GetNextEvent function, plus 13 extended events.

Remember, the event codes are returned from the Toolbox when TaskMaster is called. You don't have to examine the What field of the Event Record, as is done with GetNextEvent, to determine which event took place.

The 13 extra values, or extended event codes, are shown in Table E-6.

**Table E-6. Extended Event Codes**

| Event Code | Description |
|---|---|
| 16 | Mouse is in desk. |
| 17 | A menu item was selected. |
| 18 | Mouse is in the system window. |
| 19 | Mouse is in the content of a window. |
| 20 | Mouse is in drag. |
| 21 | Mouse is in grow. |
| 22 | Mouse is in goaway. |
| 23 | Mouse is in zoom. |
| 24 | Mouse is in info bar. |
| 25 | Mouse is in vertical scroll. |
| 26 | Mouse is in horizontal scroll. |
| 27 | Mouse is in frame. |
| 28 | Mouse is in drop. |

**TaskData.** The two extra fields on the event record help to further describe the above codes. TaskData contains additional information about the extended event code. For the standard event codes 0–15, TaskData will be blank. But for the extended event codes 16–28, Task Data contains the values shown in Table E-7.

**Table E-7. Meaning of TaskData**

| Code | TaskData Values |
|---|---|
| 16 | Not used |
| 17 | Not used |
| 18 | Not used |
| 19 | Not used |
| 20 | HOW = Menu ID, LOW = $0000 |





| Code | TaskData Values |
|---|---|
| 21 | HOW = Menu ID, LOW = Menu Item |
| 22 | Window pointer |
| 23 | Window pointer |
| 24 | Window pointer |
| 25 | Window pointer |
| 26 | Window pointer |
| 27 | Window pointer |
| 28 | Window pointer |

See examples from Chapters 8 and 9 on how this field is used.

**TaskMask.** The TaskMask field is similar to the event mask. It's used to filter out certain types of events monitored by the TaskMaster. These events are above and beyond those already filtered by the event mask. Both an event mask and a TaskMask are required by TaskMaster.

By setting specific bits in the TaskMask, your program can direct TaskMaster to return only the results of specific events. Table E-8 shows which bits in the TaskMask field affect which events. Note that bits 13–31 must always be set to 0, or an error results.

**Table E-8. Bits in TaskMask**

| Bit | TaskMaster Scans for, if Set |
|---|---|
| 0 | MenuKey: menu item key equivalents |
| 1 | Update handling |
| 2 | FindWindow: mouse click in a window |
| 3 | MenuSelect: choosing a menu item |
| 4 | OpenNDA: new desk accessories in the Apple menu |
| 5 | System click |
| 6 | Drag window |
| 7 | Select window |
| 8 | Track goaway button |
| 9 | Track zoom button |
| 10 | Grow window |
| 11 | Allow scrolling |
| 12 | Handle special menu items |
| 13–31 | Must be set to 0 |

It's generally a good idea to set all the important bits. When this field is set to a value of $000003FFF, it will scan for and be able to handle all conceivable events.

## Appendix F

# QuickDraw II Color Information

In the current version of the Apple IIGS, the color tables used by QuickDraw II are stored at the following addresses. Each color table is $20 bytes long. (These address may change with future releases of the Apple IIGS ROMs):

**Table F-1. Color Table Locations**

| Color Table | Address |
|---|---|
| 0 | $E19E00 |
| 1 | $E19E20 |
| 2 | $E19E40 |
| 3 | $E19E60 |
| 4 | $E19E80 |
| 5 | $E19EA0 |
| 6 | $E19EC0 |
| 7 | $E19EE0 |
| 8 | $E19F00 |
| 9 | $E19F20 |
| 10 | $E19F40 |
| 11 | $E19F60 |
| 12 | $E19F80 |
| 13 | $E19FA0 |
| 14 | $E19FC0 |
| 15 | $E19FE0 |

**Colors in the 320 mode.** In the 320 mode, nibble positions for each color are as follows:

**Table F-2. Color Nibble Positions**

| Color Value | Low Intensity | High Intensity |
|---|---|---|
| Blue | $0001 | $000F |
| Green | $0010 | $00F0 |
| Red | $0100 | $0F00 |

A color value of $0000 is black (all three colors are turned off). A color value of $0FFF is white (all three colors are at their highest intensity). Note how each color has 16 steps of intensity (from $0 to $F).





**Table F-3. Standard Color Table in 320 Mode**

| Color Value | Color Number | Setting |
|---|---|---|
| Black | 0 | $0000 |
| Dark Gray | 1 | $0777 |
| Brown | 2 | $0841 |
| Purple | 3 | $07C2 |
| Blue | 4 | $000F |
| Dark Green | 5 | $0080 |
| Orange | 6 | $0F70 |
| Red | 7 | $0D00 |
| Beige | 8 | $0FA9 |
| Yellow | 9 | $0FF0 |
| Green | 10 | $00E0 |
| Light Blue | 11 | $04DF |
| Lilac | 12 | $0DAF |
| Periwinkle | 13 | $078F |
| Light Gray | 14 | $0CCC |
| White | 15 | $0FFF |

**Colors in the 640 mode.** In the 640 mode, nibble positions for each color are as follows:

**Table F-4. Color Nibble Positions**

| Color | Value |
|---|---|
| Blue | $000F |
| Green | $00F0 |
| Red | $0F00 |

Unlike the 320 mode, there are only two values for each color in the 640 mode: $0 for off and $F for on.

**Table F-5. Standard Color Table in 640 Mode**

| Color Value | Color Number | Setting |
|---|---|---|
| Black | 0 | $0000 |
| Red | 1 | $0F00 |
| Green | 2 | $00F0 |
| White | 3 | $0FFF |
| Black | 4 | $0000 |
| Blue | 5 | $000F |
| Yellow | 6 | $0FF0 |
| White | 7 | $0FFF |
| Black | 8 | $0000 |
| Red | 9 | $0F00 |

## Appendix F

| Color Value | Color Number | Setting |
|---|---|---|
| Green | 10 | $00F0 |
| White | 11 | $0FFF |
| Black | 12 | $0000 |
| Blue | 13 | $000F |
| Yellow | 14 | $0FF0 |
| White | 15 | $0FFF |



# Index

# Programming the Apple IIGS Toolbox

*Advanced Programming Techniques for the Apple IIGS Toolbox* provides you with the golden key to programming the Apple IIGS. Rare is the programming book that can claim to be both solidly packed with information and well written. This is one of those books.

Written by recognized IIGS authorities Morgan Davis and Dan Gookin, *Advanced Programming Techniques for the Apple IIGS Toolbox* delves into the intricacies of the powerful set of libraries known collectively as the Toolbox. You'll appreciate the equal treatment given to machine language, Pascal, and C and the organization that allows you to come to the book with a specific problem and leave with a realistic solution.

Here are just a few things you'll find in *Advanced Programming Techniques for the Apple IIGS Toolbox*:

- Many programming examples
- DeskTop programs
- Icons
- Interrupts
- ProDOS 16
- Human Interface Guidelines

*Advanced Programming Techniques for the Apple IIGS Toolbox* is the library of vital programming data that will earn its place next to your IIGS time and again.

$19.95

ISBN 0-87455-130-7

51995

9 780874 551303